The Art of Unix Programming

by Eric Steven Raymond

The Art of Unix Programming

by Eric Steven Raymond

Dedication

To Ken Thompson and Dennis Ritchie, because you inspired me.

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

Unix is not so much an operating system as an oral history.
--
<author>NealStephenson</author>

There is a vast difference between knowledge and expertise. Knowledge lets you deduce the right thing to do; expertise makes the right thing a reflex, hardly requiring conscious thought at all.

This book has a lot of knowledge in it, but it is mainly about expertise. It is going to try to teach you the things about Unix development that Unix experts know, but aren't aware that they know. It is therefore less about technicalia and more about *shared culture* than most Unix books — both explicit and implicit culture, both conscious and unconscious traditions. It is not a 'how-to' book, it is a 'why-to' book.

The why-to has great practical importance, because far too much software is poorly designed. Much of it suffers from bloat, is exceedingly hard to maintain, and is too difficult to port to new platforms or extend in ways the original programmers didn't anticipate. These problems are symptoms of bad design. We hope that readers of this book will learn something of what Unix has to teach about good design.

This book is divided into four parts: Context, Design, Tools, and Community. The first part (Context) is philosophy and history, to help provide foundation and motivation for what follows. The second part (Design) unfolds the principles of the Unix philosophy into more specific advice about design and implementation. The third part (Tools) focuses on the software Unix provides for helping you solve problems. The fourth part (Community) is about the human-to-human transactions and agreements that make the Unix culture so effective at what it does.

Because this is a book about shared culture, I never planned to write it alone. You will notice that the text includes guest appearances by prominent Unix developers, the shapers of the Unix tradition. The book went through an extended public review process during which I invited these luminaries to comment on and argue with the text. Rather than submerging the results of that review process in the final version, these guests were encouraged to speak with their own voices, amplifying and developing and even disagreeing with the main line of the text.

In this book, when I use the editorial 'we' it is not to pretend omniscience but to reflect the fact that it attempts to articulate the expertise of an entire community.

Because this book is aimed at transmitting culture, it includes much more in the way of history and folklore and asides than is normal for a technical book. Enjoy; these things, too, are part of your education as a Unix programmer. No single one of the historical details is vital, but the gestalt of them all is important. We think it makes a more interesting story this way. More importantly, understanding where Unix came from and how it got the way it is will help you develop an intuitive feel for the Unix style.

For the same reason, we refuse to write as if history is over. You will find an unusually large number of references to the time of writing in this book. We do not wish to pretend that current practice reflects some sort of timeless and perfectly logical outcome of preordained destiny. References to time of writing are meant as an alert to the reader two or three or five years hence that the associated statements of fact may have become dated and should be double-checked.

Other things this book is not is neither a C tutorial, nor a guide to the Unix commands and API. It is not a reference for *sed* or *yacc* or Perl or Python. It's not a network programming primer, nor an exhaustive guide to the mysteries of X. It's not a tour of Unix's internals and architecture, either. Other books cover these specifics better, and this book points you at them as appropriate.

Beyond all these technical specifics, the Unix culture has an unwritten engineering tradition that has developed over literally millions of man-years[1] of skilled effort. This book is written in the belief that understanding that tradition, and adding its design patterns to your toolkit, will help you become a better programmer and designer.

Cultures consist of people, and the traditional way to learn Unix culture is from other people and through the folklore, by osmosis. This book is not a substitute for person-to-person acculturation, but it can help accelerate the process by allowing you to tap the experience of others.

Who Should Read This Book

You should read this book if you are an experienced Unix programmer who is often in the position of either educating novice programmers or debating partisans of other operating systems, and you find it hard to articulate the benefits of the Unix approach.

You should read this book if you are a C, C++, or Java programmer with experience on other operating systems and you are about to start a Unix-based project.

[1]The three and a half decades between 1969 and 2003 is a long time. Going by the historical trend curve in number of Unix sites during that period, probably somewhere upwards of fifty million man-years have been plowed into Unix development worldwide.

You should read this book if you are a Unix user with novice-level up to middle-level skills in the operating system, but little development experience, and want to learn how to design software effectively under Unix.

You should read this book if you are a non-Unix programmer who has figured out that the Unix tradition might have something to teach you. We believe you're right, and that the Unix philosophy can be exported to other operating systems. So we will pay more attention to non-Unix environments (especially Microsoft operating systems) than is usual in a Unix book; and when tools and case studies are portable, we say so.

You should read this book if you are an application architect considering platforms or implementation strategies for a major general-market or vertical application. It will help you understand the strengths of Unix as a development platform, and of the Unix tradition of open source as a development method.

You should *not* read this book if what you are looking for is the details of C coding or how to use the Unix kernel API. There are many good books on these topics; *Advanced Programming in the Unix Environment* [Stevens92] is classic among explorations of the Unix API, and *The Practice of Programming* [Kernighan-Pike99] is recommended reading for all C programmers (indeed for all programmers in any language).

How to Use This Book

This book is both practical and philosophical. Some parts are aphoristic and general, others will examine specific case studies in Unix development. We will precede or follow general principles and aphorisms with examples that illustrate them: examples drawn not from toy demonstration programs but rather from real working code that is in use every day.

We have deliberately avoided filling the book with lots of code or specification-file examples, even though in many places this might have made it easier to write (and in some places perhaps easier to read!). Most books about programming give too many low-level details and examples, but fail at giving the reader a high-level feel for what is really going on. In this book, we prefer to err in the opposite direction.

Therefore, while you will often be invited to read code and specification files, relatively few are actually included in the book. Instead, we point you at examples on the Web.

Absorbing these examples will help solidify the principles you learn into semi-instinctive working knowledge. Ideally, you should read this book near the console of a running Unix system, with a Web browser handy. Any Unix will do, but the software case studies are more likely to be preinstalled and immediately available for inspection on a Linux system. The pointers in the book are invitations to browse and experiment. Introduction of these pointers is paced so that wandering off to explore for a while won't break up exposition that has to be continuous.

Note: While we have made every effort to cite URLs that should remain stable and usable, there is no way we can guarantee this. If you find that a cited link has gone stale, use common sense and do a phrase search with your favorite Web search engine. Where possible we suggest ways to do this near the URLs we cite.

Most abbreviations used in this book are expanded at first use. For convenience, we have also provided a glossary in an appendix.

References are usually by author name. Numbered footnotes are for URLs that would intrude on the text or that we suspect might be perishable; also for asides, war stories, and jokes.[2]

To make this book more accessible to less technical readers, we invited some non-programmers to read it and identify terms that seemed both obscure and necessary to the flow of exposition. We also use footnotes for definitions of elementary terms that an experienced programmer is unlikely to need.

Related References

Some famous papers and a few books by Unix's early developers have mined this territory before. Kernighan and Pike's *The Unix Programming Environment* [Kernighan-Pike84] stands out among these and is rightly considered a classic. But today it shows its age a bit; it doesn't cover the Internet, and the World Wide Web or the new wave of interpreted languages like Perl, Tcl, and Python.

About halfway into the composition of this book, we learned of Mike Gancarz's *The Unix Philosophy* [Gancarz]. This book is excellent within its range, but did not attempt to cover the full spectrum of topics we felt needed to be addressed. Nevertheless we are grateful to the author for the reminder that the very simplest Unix design patterns have been the most persistent and successful ones.

[2]This particular footnote is dedicated to Terry Pratchett, whose use of footnotes is quite...inspiring.

The Pragmatic Programmer [Hunt-Thomas] is a witty and wise disquisition on good design practice pitched at a slightly different level of the software-design craft (more about coding, less about higher-level partitioning of problems) than this book. The authors' philosophy is an outgrowth of Unix experience, and it is an excellent complement to this book.

The Practice of Programming [Kernighan-Pike99] covers some of the same ground as *The Pragmatic Programmer* from a position deep within the Unix tradition.

Finally (and with admitted intent to provoke) we recommend *Zen Flesh, Zen Bones* [Reps-Senzaki], an important collection of Zen Buddhist primary sources. References to Zen are scattered throughout this book. They are included because Zen provides a vocabulary for addressing some ideas that turn out to be very important for software design but are otherwise very difficult to hold in the mind. Readers with religious attachments are invited to consider Zen not as a religion but as a therapeutic form of mental discipline — which, in its purest non-theistic forms, is exactly what Zen is.

Conventions Used in This Book

The term "UNIX" is technically and legally a trademark of The Open Group, and should formally be used only for operating systems which are certified to have passed The Open Group's elaborate standards-conformance tests. In this book we use "Unix" in the looser sense widely current among programmers, to refer to any operating system (whether formally Unix-branded or not) that is either genetically descended from Bell Labs's ancestral Unix code or written in close imitation of its descendants. In particular, Linux (from which we draw most of our examples) is a Unix under this definition.

This book employs the Unix manual page convention of tagging Unix facilities with a following manual section in parentheses, usually on first introduction when we want to emphasize that this is a Unix command. Thus, for example, read "munger(1)" as "the 'munger' program, which will be documented in section 1 (user tools) of the Unix manual pages, if it's present on your system". Section 2 is C system calls, section 3 is C library calls, section 5 is file formats and protocols, section 8 is system administration tools. Other sections vary among Unixes but are not cited in this book. For more, type **man 1 man** at your Unix shell prompt (older System V Unixes may require **man -s 1 man**).

Sometimes we mention a Unix application (such as *Emacs*, without a manual-section suffix and capitalized. This is a clue that the name actually represents a well-established family of Unix

programs with essentially the same function, and we are discussing generic properties of all of them. *Emacs*, for example, includes *xemacs*.

At various points later in this book we refer to ‘old school’ and ‘new school’ methods. As with rap music, new-school starts about 1990. In this context, it’s associated with the rise of scripting languages, GUIs, open-source Unixes, and the Web. Old-school refers to the pre-1990 (and especially pre-1985) world of expensive (shared) computers, proprietary Unixes, scripting in shell, and C everywhere. This difference is worth pointing out because cheaper and less memory-constrained machines have wrought some significant changes on the Unix programming style.

Our Case Studies

A lot of books on programming rely on toy examples constructed specifically to prove a point. This one won’t. Our case studies will be real, pre-existing pieces of software that are in production use every day. Here are some of the major ones:

cdrtools/*xcdroast*
: These two separate projects are usually used together. The *cdrtools* package is a set of CLI tools for writing CD-ROMs; Web search for “cdrtools”. The *xcdroast* application is a GUI front end for cdrtools; see the xcdroast project site [http://www.xcdroast.org/].

fetchmail
: The *fetchmail* program retrieves mail from remote-mail servers using the POP3 or IMAP post-office protocols. See the fetchmail home page [http://www.catb.org/~esr/fetchmail] (or search for “fetchmail” on the Web).

GIMP
: The *GIMP* (GNU Image Manipulation Program) is a full-featured paint, draw, and image-manipulation program that can edit a huge variety of graphical formats in sophisticated ways. Sources are available from the GIMP home page [http://www.gimp.org/] (or search for "GIMP" on the Web).

mutt — The *mutt* mail user agent is the current best-of-breed among text-based Unix electronic mail agents, with notably good support for MIME (Multipurpose Internet Mail Extensions) and the use of privacy aids such as PGP (Pretty Good Privacy) and GPG (GNU Privacy Guard). Source code and executable binaries are available at the Mutt project site [http://www.mutt.org].

xmlto — The *xmlto* command renders DocBook and other XML documents in various output formats, including HTML and text and PostScript. For sources and documentation, see the xmlto project site [http://cyberelk.net/tim/xmlto/].

To minimize the amount of code the user needs to read to understand the examples, we have tried to choose case studies that can be used more than once, ideally to illustrate several different design principles and practices. For this same reason, many of the examples are from my projects. No claim that these are the best possible ones is implied, merely that I find them sufficiently familiar to be useful for multiple expository purposes.

Author’s Acknowledgements

The guest contributors (Ken Arnold, Steven M. Bellovin, Stuart Feldman, Jim Gettys, Steve Johnson, Brian Kernighan, David Korn, Mike Lesk, Doug McIlroy, Marshall Kirk McKusick, Keith Packard, Henry Spencer, and Ken Thompson) added a great deal of value to this book. Doug McIlroy, in particular, went far beyond the call of duty in the thoroughness of his critique and the depth of his contributions, displaying the same care and dedication to excellence which he brought to managing the original Unix research group thirty years ago.

Special thanks go to Rob Landley and to my wife Catherine Raymond, both of whom delivered intensive line-by-line critiques of manuscript drafts. Rob’s insightful and attentive commentary actually inspired more than one entire chapter in the final manuscript, and he had a lot to do with its present organization and range; if he had written all the text he pushed me to improve, I would have to call him a co-author. Cathy was my test audience representing non-technical readers; to the extent this book is accessible to people who aren’t already programmers, that’s largely her doing.

This book benefited from discussions with many other people over the five years it took me to write it. Mark M. Miller helped me achieve enlightenment about threads. John Cowan supplied some insights about interface design patterns and drafted the case studies of *wily* and VM/CMS, and Jef

Raskin showed me where the Rule of Least Surprise comes from. The UIUC System Architecture Group contributed useful feedback on early chapters. The sections on *What Unix Gets Wrong* and *Flexibility in Depth* were directly inspired by their review. Russell J. Nelson contributed the material on Bernstein chaining in Chapter 7. Jay Maynard contributed most of the material in the MVS case study in Chapter 3. Les Hatton provided many helpful comments on the *Languages* chapter and motivated the portion of Chapter 4 on *Optimal Module Size*. David A. Wheeler contributed many perceptive criticisms and some case-study material, especially in the Design part. Russ Cox helped develop the survey of Plan 9. Dennis Ritchie corrected me on some historical points about C.

Hundreds of Unix programmers, far too many to list here, contributed advice and comments during the book's public review period between January and June of 2003. As always, I found the process of open peer review over the Web both intensely challenging and intensely rewarding. Also as always, responsibility for any errors in the resulting work remains my own.

The expository style and some of the concerns of this book have been influenced by the design patterns movement; indeed, I flirted with the idea of titling the book *Unix Design Patterns*. I didn't, because I disagree with some of the implicit central dogmas of the movement and don't feel the need to use all its formal apparatus or accept its cultural baggage. Nevertheless, my approach has certainly been influenced by Christopher Alexander's work[3] (especially *The Timeless Way of Building* and *A Pattern Language*), and I owe the Gang of Four and other members of their school a large debt of gratitude for showing me how it is possible to use Alexander's insights to talk about software design at a high level without merely uttering vague and useless generalities. Interested readers should see *Design Patterns: Elements of Reusable Object-Oriented Software* [GangOfFour] for an introduction to design patterns.

The title of this book is, of course, a reference to Donald Knuth's *The Art of Computer Programming*. While not specifically associated with the Unix tradition, Knuth has been an influence on us all.

Editors with vision and imagination aren't as common as they should be. Mark Taub is one; he saw merit in a stalled project and skillfully nudged me into finishing it. Copy editors with a good ear for prose style and enough ability to improve writing that isn't like theirs are even less common, but Mary Lou Nohr makes that grade. Jerry Votta seized on my concept for the cover and made it look better than I had imagined. The whole crew at Addison-Wesley gets high marks for making the editorial and production process as painless as possible, and for cheerfully accommodating my control-freak tendencies not just over the text but deep into the details of the book's visual design, art, and marketing.

[3]An appreciation of Alexander's work, with links to on-line versions of significant portions, may be found at Some Notes on Christopher Alexander [http://www.math.utsa.edu/sphere/salingar/Chris.text.html].

Part I. Context

Chapter 1. Philosophy

Philosophy Matters

Those who do not understand Unix are condemned to reinvent it, poorly.
--
<author>HenrySpencer</author>
Usenet signature, November 1987

Culture? What Culture?

This is a book about Unix programming, but in it we're going to toss around the words 'culture', 'art', and 'philosophy' a lot. If you are not a programmer, or you are a programmer who has had little contact with the Unix world, this may seem strange. But Unix has a culture; it has a distinctive art of programming; and it carries with it a powerful design philosophy. Understanding these traditions will help you build better software, even if you're developing for a non-Unix platform.

Every branch of engineering and design has technical cultures. In most kinds of engineering, the unwritten traditions of the field are parts of a working practitioner's education as important as (and, as experience grows, often more important than) the official handbooks and textbooks. Senior engineers develop huge bodies of implicit knowledge, which they pass to their juniors by (as Zen Buddhists put it) "a special transmission, outside the scriptures".

Software engineering is generally an exception to this rule; technology has changed so rapidly, software environments have come and gone so quickly, that technical cultures have been weak and ephemeral. There are, however, exceptions to this exception. A very few software technologies have proved durable enough to evolve strong technical cultures, distinctive arts, and an associated design philosophy transmitted across generations of engineers.

The Unix culture is one of these. The Internet culture is another — or, in the twenty-first century, arguably the same one. The two have grown increasingly difficult to separate since the early 1980s, and in this book we won't try particularly hard.

The Durability of Unix

Unix was born in 1969 and has been in continuous production use ever since. That's several geologic eras by computer-industry standards — older than the PC or workstations or microprocessors or

even video display terminals, and contemporaneous with the first semiconductor memories. Of all production timesharing systems today, only IBM's VM/CMS can claim to have existed longer, and Unix machines have provided hundreds of thousands of times more service hours; indeed, Unix has probably supported more computing than all other timesharing systems put together.

Unix has found use on a wider variety of machines than any other operating system can claim. From supercomputers to handhelds and embedded networking hardware, through workstations and servers and PCs and minicomputers, Unix has probably seen more architectures and more odd hardware than any three other operating systems combined.

Unix has supported a mind-bogglingly wide spectrum of uses. No other operating system has shone simultaneously as a research vehicle, a friendly host for technical custom applications, a platform for commercial-off-the-shelf business software, and a vital component technology of the Internet.

Confident predictions that Unix would wither away, or be crowded out by other operating systems, have been made yearly since its infancy. And yet Unix, in its present-day avatars as Linux and BSD and Solaris and MacOS X and half a dozen other variants, seems stronger than ever today.

> Robert Metcalf [the inventor of Ethernet] says that if something comes along to replace Ethernet, it will be called "Ethernet", so therefore Ethernet will never die.[4] Unix has already undergone several such transformations.
>
> —
>
> <author>KenThompson</author>

At least one of Unix's central technologies — the C language — has been widely naturalized elsewhere. Indeed it is now hard to imagine doing software engineering without C as a ubiquitous common language of systems programming. Unix also introduced both the now-ubiquitous tree-shaped file namespace with directory nodes and the pipeline for connecting programs.

Unix's durability and adaptability have been nothing short of astonishing. Other technologies have come and gone like mayflies. Machines have increased a thousandfold in power, languages have mutated, industry practice has gone through multiple revolutions — and Unix hangs in there, still producing, still paying the bills, and still commanding loyalty from many of the best and brightest software technologists on the planet.

[4] In fact, Ethernet has already been replaced by a different technology with the same name — twice. Once when coax was replaced with twisted pair, and a second time when gigabit Ethernet came in.

One of the many consequences of the exponential power-versus-time curve in computing, and the corresponding pace of software development, is that 50% of what one knows becomes obsolete over every 18 months. Unix does not abolish this phenomenon, but does do a good job of containing it. There's a bedrock of unchanging basics — languages, system calls, and tool invocations — that one can actually keep using for years, even decades. Elsewhere it is impossible to predict what will be stable; even entire operating systems cycle out of use. Under Unix, there is a fairly sharp distinction between transient knowledge and lasting knowledge, and one can know ahead of time (with about 90% certainty) which category something is likely to fall in when one learns it. Thus the loyalty Unix commands.

Much of Unix's stability and success has to be attributed to its inherent strengths, to design decisions Ken Thompson, Dennis Ritchie, Brian Kernighan, Doug McIlroy, Rob Pike and other early Unix developers made back at the beginning; decisions that have been proven sound over and over. But just as much is due to the design philosophy, art of programming, and technical culture that grew up around Unix in the early days. This tradition has continuously and successfully propagated itself in symbiosis with Unix ever since.

The Case against Learning Unix Culture

Unix's durability and its technical culture are certainly of interest to people who already like Unix, and perhaps to historians of technology. But Unix's original application as a general-purpose timesharing system for mid-sized and larger computers is rapidly receding into the mists of history, killed off by personal workstations. And there is certainly room for doubt that it will ever achieve success in the mainstream business-desktop market now dominated by Microsoft.

Outsiders have frequently dismissed Unix as an academic toy or a hacker's sandbox. One well-known polemic, the *Unix Hater's Handbook* [Garfinkel], follows an antagonistic line nearly as old as Unix itself in writing its devotees off as a cult religion of freaks and losers. Certainly the colossal and repeated blunders of AT&T, Sun, Novell, and other commercial vendors and standards consortia in mispositioning and mismarketing Unix have become legendary.

Even from within the Unix world, Unix has seemed to be teetering on the brink of universality for so long as to raise the suspicion that it will never actually get there. A skeptical outside observer's conclusion might be that Unix is too useful to die but too awkward to break out of the back room; a perpetual niche operating system.

What confounds the skeptics' case is, more than anything else, the rise of Linux and other open-source Unixes (such as the modern BSD variants). Unix's culture proved too vital to be smothered

even by a decade of vendor mismanagement. Today the Unix community itself has taken control of the technology and marketing, and is rapidly and visibly solving Unix's problems (in ways we'll examine in more detail in Chapter 20).

What Unix Gets Wrong

For a design that dates from 1969, it is remarkably difficult to identify design choices in Unix that are unequivocally wrong. There are several popular candidates, but each is still a subject of spirited debate not merely among Unix fans but across the wider community of people who think about and design operating systems.

Unix files have no structure above byte level. File deletion is irrevocable. The Unix security model is arguably too primitive. Job control is botched. There are too many different kinds of names for things. Having a file system at all may have been the wrong choice. We will discuss these technical issues in Chapter 20.

But perhaps the most enduring objections to Unix are consequences of a feature of its philosophy first made explicit by the designers of the X windowing system. X strives to provide "mechanism, not policy", supporting an extremely general set of graphics operations and deferring decisions about toolkits and interface look-and-feel (the policy) up to application level. Unix's other system-level services display similar tendencies; final choices about behavior are pushed as far toward the user as possible. Unix users can choose among multiple shells. Unix programs normally provide many behavior options and sport elaborate preference facilities.

This tendency reflects Unix's heritage as an operating system designed primarily for technical users, and a consequent belief that users know better than operating-system designers what their own needs are.

> This tenet was firmly established at Bell Labs by Dick Hamming[5] who insisted in the 1950s when computers were rare and expensive, that open-shop computing, where customers wrote their own programs, was imperative, because "it is better to solve the right problem the wrong way than the wrong problem the right way".
>
> —
>
> <author>DougMcIlroy</author>

[5] Yes, the Hamming of 'Hamming distance' and 'Hamming code'.

But the cost of the mechanism-not-policy approach is that when the user *can* set policy, the user *must* set policy. Nontechnical end-users frequently find Unix's profusion of options and interface styles overwhelming and retreat to systems that at least pretend to offer them simplicity.

In the short term, Unix's laissez-faire approach may lose it a good many nontechnical users. In the long term, however, it may turn out that this 'mistake' confers a critical advantage — because policy tends to have a short lifetime, mechanism a long one. Today's fashion in interface look-and-feel too often becomes tomorrow's evolutionary dead end (as people using obsolete X toolkits will tell you with some feeling!). So the flip side of the flip side is that the "mechanism, not policy" philosophy may enable Unix to renew its relevance long after competitors more tied to one set of policy or interface choices have faded from view.[6]

What Unix Gets Right

The explosive recent growth of Linux, and the increasing importance of the Internet, give us good reasons to suppose that the skeptics' case is wrong. But even supposing the skeptical assessment is true, Unix culture is worth learning because there are some things that Unix and its surrounding culture clearly do better than any competitors.

Open-Source Software

Though the term "open source" and the Open Source Definition were not invented until 1998, peer-review-intensive development of freely shared source code was a key feature of the Unix culture from its beginnings.

For its first ten years AT&T's original Unix, and its primary variant Berkeley Unix, were normally distributed with source code. This enabled most of the other good things that follow here.

Cross-Platform Portability and Open Standards

Unix is still the only operating system that can present a consistent, documented application programming interface (API) across a heterogeneous mix of computers, vendors, and special-purpose hardware. It is the only operating system that can scale from embedded chips and handhelds,

[6]Jim Gettys, one of the architects of X (and a contributor to this book), has meditated in depth on how X's laissez-faire style might be productively carried forward in *The Two-Edged Sword* [Gettys]. This essay is well worth reading, both for its specific proposals and for its expression of the Unix mindset.

up through desktop machines, through servers, and all the way to special-purpose number-crunching behemoths and database back ends.

The Unix API is the closest thing to a hardware-independent standard for writing truly portable software that exists. It is no accident that what the IEEE originally called the *Portable Operating System Standard* quickly got a suffix added to its acronym and became POSIX. A Unix-equivalent API was the only credible model for such a standard.

Binary-only applications for other operating systems die with their birth environments, but Unix sources are forever. Forever, at least, given a Unix technical culture that polishes and maintains them across decades.

The Internet and the World Wide Web

The Defense Department's contract for the first production TCP/IP stack went to a Unix development group because the Unix in question was largely open source. Besides TCP/IP, Unix has become the one indispensable core technology of the Internet Service Provider industry. Ever since the demise of the TOPS family of operating systems in the mid-1980s, most Internet server machines (and effectively all above the PC level) have relied on Unix.

Not even Microsoft's awesome marketing clout has been able to dent Unix's lock on the Internet. While the TCP/IP standards (on which the Internet is based) evolved under TOPS-10 and are theoretically separable from Unix, attempts to make them work on other operating systems have been bedeviled by incompatibilities, instabilities, and bugs. The theory and specifications are available to anyone, but the engineering tradition to make them into a solid and working reality exists only in the Unix world.[7]

The Internet technical culture and the Unix culture began to merge in the early 1980s, and are now inseparably symbiotic. The design of the World Wide Web, the modern face of the Internet, owes as much to Unix as it does to the ancestral ARPANET. In particular, the concept of the Uniform Resource Locator (URL) so central to the Web is a generalization of the Unix idea of one uniform file namespace everywhere. To function effectively as an Internet expert, an understanding of Unix and its culture are indispensable.

The Open-Source Community

[7] Other operating systems have generally copied or cloned Unix TCP/IP implementations. It is their loss that they have not generally adopted the robust tradition of peer review that goes with it, exemplified by documents like RFC 1025 (*TCP and IP Bake Off*).

The community that originally formed around the early Unix source distributions never went away — after the great Internet explosion of the early 1990s, it recruited an entire new generation of eager hackers on home machines.

Today, that community is a powerful support group for all kinds of software development. High-quality open-source development tools abound in the Unix world (we'll examine many in this book). Open-source Unix applications are usually equal to, and are often superior to, their proprietary equivalents [Fuzz]. Entire Unix operating systems, with complete toolkits and basic applications suites, are available for free over the Internet. Why code from scratch when you can adapt, reuse, recycle, and save yourself 90% of the work?

This tradition of code-sharing depends heavily on hard-won expertise about how to make programs cooperative and reusable. And not by abstract theory, but through a lot of engineering practice — unobvious design rules that allow programs to function not just as isolated one-shot solutions but as synergistic parts of a toolkit. A major purpose of this book is to elucidate those rules.

Today, a burgeoning open-source movement is bringing new vitality, new technical approaches, and an entire generation of bright young programmers into the Unix tradition. Open-source projects including the Linux operating system and symbionts such as Apache and Mozilla have brought the Unix tradition an unprecedented level of mainstream visibility and success. The open-source movement seems on the verge of winning its bid to define the computing infrastructure of tomorrow — and the core of that infrastructure will be Unix machines running on the Internet.

Flexibility All the Way Down

Many operating systems touted as more 'modern' or 'user friendly' than Unix achieve their surface glossiness by locking users and developers into one interface policy, and offer an application-programming interface that for all its elaborateness is rather narrow and rigid. On such systems, tasks the designers have anticipated are very easy — but tasks they have not anticipated are often impossible or at best extremely painful.

Unix, on the other hand, has flexibility in depth. The many ways Unix provides to glue together programs mean that components of its basic toolkit can be combined to produce useful effects that the designers of the individual toolkit parts never anticipated.

Unix's support of multiple styles of program interface (often seen as a weakness because it increases the perceived complexity of the system to end users) also contributes to flexibility; no program

that wants to be a simple piece of data plumbing is forced to carry the complexity overhead of an elaborate GUI.

Unix tradition lays heavy emphasis on keeping programming interfaces relatively small, clean, and orthogonal — another trait that produces flexibility in depth. Throughout a Unix system, easy things are easy and hard things are at least possible.

Unix Is Fun to Hack

People who pontificate about Unix's technical superiority often don't mention what may ultimately be its most important strength, the one that underlies all its successes. Unix is fun to hack.

Unix boosters seem almost ashamed to acknowledge this sometimes, as though admitting they're having fun might damage their legitimacy somehow. But it's true; Unix is fun to play with and develop for, and always has been.

There are not many operating systems that anyone has ever described as 'fun'. Indeed, the friction and labor of development under most other environments has been aptly compared to kicking a dead whale down the beach.[8] The kindest adjectives one normally hears are on the order of "tolerable" or "not too painful". In the Unix world, by contrast, the operating system rewards effort rather than frustrating it. People programming under Unix usually come to see it not as an adversary to be clubbed into doing one's bidding by main effort but rather as an actual positive help.

This has real economic significance. The fun factor started a virtuous circle early in Unix's history. People liked Unix, so they built more programs for it that made it nicer to use. Today people build entire, production-quality open-source Unix systems as a hobby. To understand how remarkable this is, ask yourself when you last heard of anybody cloning OS/360 or VAX VMS or Microsoft Windows for fun.

The 'fun' factor is not trivial from a design point of view, either. The kind of people who become programmers and developers have 'fun' when the effort they have to put out to do a task challenges them, but is just within their capabilities. 'Fun' is therefore a sign of peak efficiency. Painful development environments waste labor and creativity; they extract huge hidden costs in time, money, and opportunity.

[8] This was originally said of the IBM MVS TSO facility by Stephen C. Johnson, perhaps better known as the author of *yacc*.

If Unix were a failure in every other way, the Unix engineering culture would be worth studying for the ways it keeps the fun in development — because that fun is a sign that it makes developers efficient, effective, and productive.

The Lessons of Unix Can Be Applied Elsewhere

Unix programmers have accumulated decades of experience while pioneering operating-system features we now take for granted. Even non-Unix programmers can benefit from studying that Unix experience. Because Unix makes it relatively easy to apply good design principles and development methods, it is an excellent place to learn them.

Other operating systems generally make good practice rather more difficult, but even so some of the Unix culture's lessons can transfer. Much Unix code (including all its filters, its major scripting languages, and many of its code generators) will port directly to any operating system supporting ANSI C (for the excellent reason that C itself was a Unix invention and the ANSI C library embodies a substantial chunk of Unix's services!).

Basics of the Unix Philosophy

The 'Unix philosophy' originated with Ken Thompson's early meditations on how to design a small but capable operating system with a clean service interface. It grew as the Unix culture learned things about how to get maximum leverage out of Thompson's design. It absorbed lessons from many sources along the way.

The Unix philosophy is not a formal design method. It wasn't handed down from the high fastnesses of theoretical computer science as a way to produce theoretically perfect software. Nor is it that perennial executive's mirage, some way to magically extract innovative but reliable software on too short a deadline from unmotivated, badly managed, and underpaid programmers.

The Unix philosophy (like successful folk traditions in other engineering disciplines) is bottom-up, not top-down. It is pragmatic and grounded in experience. It is not to be found in official methods and standards, but rather in the implicit half-reflexive knowledge, the *expertise* that the Unix culture transmits. It encourages a sense of proportion and skepticism — and shows both by having a sense of (often subversive) humor.

Doug McIlroy, the inventor of Unix pipes and one of the founders of the Unix tradition, had this to say at the time [McIlroy78]:

> (i) Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.
>
> (ii) Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
>
> (iii) Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
>
> (iv) Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

He later summarized it this way (quoted in *A Quarter Century of Unix* [Salus]):

> This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

Rob Pike, who became one of the great masters of C, offers a slightly different angle in *Notes on C Programming* [Pike]:

> Rule 1. You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.
>
> Rule 2. Measure. Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.
>
> Rule 3. Fancy algorithms are slow when `n` is small, and `n` is usually small. Fancy algorithms have big constants. Until you know that `n` is frequently going to be big, don't get fancy. (Even if `n` does get big, use Rule 2 first.)
>
> Rule 4. Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures.

> Rule 5. Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.[9]
>
> Rule 6. There is no Rule 6.

Ken Thompson, the man who designed and implemented the first Unix, reinforced Pike's rule 4 with a gnomic maxim worthy of a Zen patriarch:

> When in doubt, use brute force.

More of the Unix philosophy was implied not by what these elders said but by what they did and the example Unix itself set. Looking at the whole, we can abstract the following ideas:

1. Rule of Modularity: Write simple parts connected by clean interfaces.
2. Rule of Clarity: Clarity is better than cleverness.
3. Rule of Composition: Design programs to be connected to other programs.
4. Rule of Separation: Separate policy from mechanism; separate interfaces from engines.
5. Rule of Simplicity: Design for simplicity; add complexity only where you must.
6. Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.
7. Rule of Transparency: Design for visibility to make inspection and debugging easier.
8. Rule of Robustness: Robustness is the child of transparency and simplicity.
9. Rule of Representation: Fold knowledge into data so program logic can be stupid and robust.
10. Rule of Least Surprise: In interface design, always do the least surprising thing.

[9] Pike's original adds "(See Brooks p. 102.)" here. The reference is to an early edition of *The Mythical Man-Month* [Brooks]; the quote is "Show me your flow charts and conceal your tables and I shall continue to be mystified, show me your tables and I won't usually need your flow charts; they'll be obvious".

11. Rule of Silence: When a program has nothing surprising to say, it should say nothing.
12. Rule of Repair: When you must fail, fail noisily and as soon as possible.
13. Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.
14. Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.
15. Rule of Optimization: Prototype before polishing. Get it working before you optimize it.
16. Rule of Diversity: Distrust all claims for “one true way”.
17. Rule of Extensibility: Design for the future, because it will be here sooner than you think.

If you’re new to Unix, these principles are worth some meditation. Software-engineering texts recommend most of them; but most other operating systems lack the right tools and traditions to turn them into practice, so most programmers can’t apply them with any consistency. They come to accept blunt tools, bad designs, overwork, and bloated code as normal — and then wonder what Unix fans are so annoyed about.

Rule of Modularity: Write simple parts connected by clean interfaces.

As Brian Kernighan once observed, “Controlling complexity is the essence of computer programming” [Kernighan-Plauger]. Debugging dominates development time, and getting a working system out the door is usually less a result of brilliant design than it is of managing not to trip over your own feet too many times.

Assemblers, compilers, flowcharting, procedural programming, structured programming, “artificial intelligence”, fourth-generation languages, object orientation, and software-development methodologies without number have been touted and sold as a cure for this problem. All have failed as cures, if only because they ‘succeeded’ by escalating the normal level of program complexity to the point where (once again) human brains could barely cope. As Fred Brooks famously observed [Brooks], there is no silver bullet.

The only way to write complex software that won’t fall on its face is to hold its global complexity down — to build it out of simple parts connected by well-defined interfaces, so that most problems are local and you can have some hope of upgrading a part without breaking the whole.

Rule of Clarity: Clarity is better than cleverness.

Because maintenance is so important and so expensive, write programs as if the most important communication they do is not to the computer that executes them but to the human beings who will read and maintain the source code in the future (including yourself).

In the Unix tradition, the implications of this advice go beyond just commenting your code. Good Unix practice also embraces choosing your algorithms and implementations for future maintainability. Buying a small increase in performance with a large increase in the complexity and obscurity of your technique is a bad trade — not merely because complex code is more likely to harbor bugs, but also because complex code will be harder to read for future maintainers.

Code that is graceful and clear, on the other hand, is less likely to break — and more likely to be instantly comprehended by the next person to have to change it. This is important, especially when that next person might be yourself some years down the road.

> Never struggle to decipher subtle code three times. Once might be a one-shot fluke, but if you find yourself having to figure it out a second time — because the first was too long ago and you've forgotten details — it is time to comment the code so that the third time will be relatively painless.
>
> —
> <author>HenrySpencer</author>

Rule of Composition: Design programs to be connected with other programs.

It's hard to avoid programming overcomplicated monoliths if none of your programs can talk to each other.

Unix tradition strongly encourages writing programs that read and write simple, textual, stream-oriented, device-independent formats. Under classic Unix, as many programs as possible are written as simple *filters*, which take a simple text stream on input and process it into another simple text stream on output.

Despite popular mythology, this practice is favored not because Unix programmers hate graphical user interfaces. It's because if you don't write programs that accept and emit simple text streams, it's much more difficult to hook the programs together.

Text streams are to Unix tools as messages are to objects in an object-oriented setting. The simplicity of the text-stream interface enforces the encapsulation of the tools. More elaborate forms of inter-process communication, such as remote procedure calls, show a tendency to involve programs with each others' internals too much.

To make programs composable, make them independent. A program on one end of a text stream should care as little as possible about the program on the other end. It should be made easy to replace one end with a completely different implementation without disturbing the other.

GUIs can be a very good thing. Complex binary data formats are sometimes unavoidable by any reasonable means. But before writing a GUI, it's wise to ask if the tricky interactive parts of your program can be segregated into one piece and the workhorse algorithms into another, with a simple command stream or application protocol connecting the two. Before devising a tricky binary format to pass data around, it's worth experimenting to see if you can make a simple textual format work and accept a little parsing overhead in return for being able to hack the data stream with general-purpose tools.

When a serialized, protocol-like interface is not natural for the application, proper Unix design is to at least organize as many of the application primitives as possible into a library with a well-defined API. This opens up the possibility that the application can be called by linkage, or that multiple interfaces can be glued on it for different tasks.

(We discuss these issues in detail in Chapter 7.)

Rule of Separation: Separate policy from mechanism; separate interfaces from engines.

In our discussion of what Unix gets wrong, we observed that the designers of X made a basic decision to implement "mechanism, not policy"—to make X a generic graphics engine and leave decisions about user-interface style to toolkits and other levels of the system. We justified this by pointing out that policy and mechanism tend to mutate on different timescales, with policy changing much faster than mechanism. Fashions in the look and feel of GUI toolkits may come and go, but raster operations and compositing are forever.

Thus, hardwiring policy and mechanism together has two bad effects: It makes policy rigid and harder to change in response to user requirements, and it means that trying to change policy has a strong tendency to destabilize the mechanisms.

On the other hand, by separating the two we make it possible to experiment with new policy without breaking mechanisms. We also make it much easier to write good tests for the mechanism (policy, because it ages so quickly, often does not justify the investment).

This design rule has wide application outside the GUI context. In general, it implies that we should look for ways to separate interfaces from engines.

One way to effect that separation is, for example, to write your application as a library of C service routines that are driven by an embedded scripting language, with the application flow of control written in the scripting language rather than C. A classic example of this pattern is the *Emacs* editor, which uses an embedded Lisp interpreter to control editing primitives written in C. We discuss this style of design in Chapter 11.

Another way is to separate your application into cooperating front-end and back-end processes communicating through a specialized application protocol over sockets; we discuss this kind of design in Chapter 5 and Chapter 7. The front end implements policy; the back end, mechanism. The global complexity of the pair will often be far lower than that of a single-process monolith implementing the same functions, reducing your vulnerability to bugs and lowering life-cycle costs.

Rule of Simplicity: Design for simplicity; add complexity only where you must.

Many pressures tend to make programs more complicated (and therefore more expensive and buggy). One such pressure is technical machismo. Programmers are bright people who are (often justly) proud of their ability to handle complexity and juggle abstractions. Often they compete with their peers to see who can build the most intricate and beautiful complexities. Just as often, their ability to design outstrips their ability to implement and debug, and the result is expensive failure.

> The notion of “intricate and beautiful complexities” is almost an oxymoron. Unix programmers vie with each other for “simple and beautiful” honors — a point that’s implicit in these rules, but is well worth making overt.
>
> —
>
> <author>DougMcIlroy</author>

Even more often (at least in the commercial software world) excessive complexity comes from project requirements that are based on the marketing fad of the month rather than the reality of what customers want or software can actually deliver. Many a good design has been smothered under marketing’s pile of “checklist features” — features that, often, no customer will ever use.

And a vicious circle operates; the competition thinks it has to compete with chrome by adding more chrome. Pretty soon, massive bloat is the industry standard and everyone is using huge, buggy programs not even their developers can love.

Either way, everybody loses in the end.

The only way to avoid these traps is to encourage a software culture that knows that small is beautiful, that actively resists bloat and complexity: an engineering tradition that puts a high value on simple solutions, that looks for ways to break program systems up into small cooperating pieces, and that reflexively fights attempts to gussy up programs with a lot of chrome (or, even worse, to design programs *around* the chrome).

That would be a culture a lot like Unix's.

Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.

'Big' here has the sense both of large in volume of code and of internal complexity. Allowing programs to get large hurts maintainability. Because people are reluctant to throw away the visible product of lots of work, large programs invite overinvestment in approaches that are failed or suboptimal.

(We'll examine the issue of the right size of software in more detail in Chapter 13.)

Rule of Transparency: Design for visibility to make inspection and debugging easier.

Because debugging often occupies three-quarters or more of development time, work done early to ease debugging can be a very good investment. A particularly effective way to ease debugging is to design for *transparency* and *discoverability*.

A software system is *transparent* when you can look at it and immediately understand what it is doing and how. It is *discoverable* when it has facilities for monitoring and display of internal state so that your program not only functions well but can be *seen* to function well.

Designing for these qualities will have implications throughout a project. At minimum, it implies that debugging options should not be minimal afterthoughts. Rather, they should be designed in

from the beginning — from the point of view that the program should be able to both demonstrate its own correctness and communicate to future developers the original developer's mental model of the problem it solves.

For a program to demonstrate its own correctness, it needs to be using input and output formats sufficiently simple so that the proper relationship between valid input and correct output is easy to check.

The objective of designing for transparency and discoverability should also encourage simple interfaces that can easily be manipulated by other programs — in particular, test and monitoring harnesses and debugging scripts.

Rule of Robustness: Robustness is the child of transparency and simplicity.

Software is said to be *robust* when it performs well under unexpected conditions which stress the designer's assumptions, as well as under normal conditions.

Most software is fragile and buggy because most programs are too complicated for a human brain to understand all at once. When you can't reason correctly about the guts of a program, you can't be sure it's correct, and you can't fix it if it's broken.

It follows that the way to make robust programs is to make their internals easy for human beings to reason about. There are two main ways to do that: transparency and simplicity.

> For robustness, designing in tolerance for unusual or extremely bulky inputs is also important. Bearing in mind the Rule of Composition helps; input generated by other programs is notorious for stress-testing software (e.g., the original Unix C compiler reportedly needed small upgrades to cope well with Yacc output). The forms involved often seem useless to humans. For example, accepting empty lists/strings/etc., even in places where a human would seldom or never supply an empty string, avoids having to special-case such situations when generating the input mechanically.
>
> —
> <author>HenrySpencer</author>

One very important tactic for being robust under odd inputs is to avoid having special cases in your code. Bugs often lurk in the code for handling special cases, and in the interactions among parts of the code intended to handle different special cases.

We observed above that software is *transparent* when you can look at it and immediately see what is going on. It is *simple* when what is going on is uncomplicated enough for a human brain to reason about all the potential cases without strain. The more your programs have both of these qualities, the more robust they will be.

Modularity (simple parts, clean interfaces) is a way to organize programs to make them simpler. There are other ways to fight for simplicity. Here's another one.

Rule of Representation: Fold knowledge into data, so program logic can be stupid and robust.

Even the simplest procedural logic is hard for humans to verify, but quite complex data structures are fairly easy to model and reason about. To see this, compare the expressiveness and explanatory power of a diagram of (say) a fifty-node pointer tree with a flowchart of a fifty-line program. Or, compare an array initializer expressing a conversion table with an equivalent switch statement. The difference in transparency and clarity is dramatic. See Rob Pike's Rule 5.

Data is more tractable than program logic. It follows that where you see a choice between complexity in data structures and complexity in code, choose the former. More: in evolving a design, you should actively seek ways to shift complexity from code to data.

The Unix community did not originate this insight, but a lot of Unix code displays its influence. The C language's facility at manipulating pointers, in particular, has encouraged the use of dynamically-modified reference structures at all levels of coding from the kernel upward. Simple pointer chases in such structures frequently do duties that implementations in other languages would instead have to embody in more elaborate procedures.

(We also cover these techniques in Chapter 9.)

Rule of Least Surprise: In interface design, always do the least surprising thing.

(This is also widely known as the Principle of Least Astonishment.)

The easiest programs to use are those that demand the least new learning from the user — or, to put it another way, the easiest programs to use are those that most effectively connect to the user's pre-existing knowledge.

Therefore, avoid gratuitous novelty and excessive cleverness in interface design. If you're writing a calculator program, '+' should always mean addition! When designing an interface, model it on the interfaces of functionally similar or analogous programs with which your users are likely to be familiar.

Pay attention to your expected audience. They may be end users, they may be other programmers, or they may be system administrators. What is least surprising can differ among these groups.

Pay attention to tradition. The Unix world has rather well-developed conventions about things like the format of configuration and run-control files, command-line switches, and the like. These traditions exist for a good reason: to tame the learning curve. Learn and use them.

(We'll cover many of these traditions in Chapter 5 and Chapter 10.)

> The flip side of the Rule of Least Surprise is to avoid making things superficially similar but really a little bit different. This is extremely treacherous because the seeming familiarity raises false expectations. It's often better to make things distinctly different than to make them *almost* the same.
>
> —
>
> <author>HenrySpencer</author>

Rule of Silence: When a program has nothing surprising to say, it should say nothing.

One of Unix's oldest and most persistent design rules is that when a program has nothing interesting or surprising to say, it should *shut up*. Well-behaved Unix programs do their jobs unobtrusively, with a minimum of fuss and bother. Silence is golden.

This "silence is golden" rule evolved originally because Unix predates video displays. On the slow printing terminals of 1969, each line of unnecessary output was a serious drain on the user's time. That constraint is gone, but excellent reasons for terseness remain.

> I think that the terseness of Unix programs is a central feature of the style. When your program's output becomes another's input, it should be easy to pick out the needed bits. And for people it is a human-factors necessity — important information should not be mixed in with verbosity about internal program behavior. If all displayed information is important, important information is easy to find.
>
> —
> <author>KenArnold</author>

Well-designed programs treat the user's attention and concentration as a precious and limited resource, only to be claimed when necessary.

(We'll discuss the Rule of Silence and the reasons for it in more detail at the end of Chapter 11.)

Rule of Repair: Repair what you can — but when you must fail, fail noisily and as soon as possible.

Software should be transparent in the way that it fails, as well as in normal operation. It's best when software can cope with unexpected conditions by adapting to them, but the worst kinds of bugs are those in which the repair doesn't succeed and the problem quietly causes corruption that doesn't show up until much later.

Therefore, write your software to cope with incorrect inputs and its own execution errors as gracefully as possible. But when it cannot, make it fail in a way that makes diagnosis of the problem as easy as possible.

Consider also Postel's Prescription:[10] "Be liberal in what you accept, and conservative in what you send". Postel was speaking of network service programs, but the underlying idea is more general. Well-designed programs cooperate with other programs by making as much sense as they can from ill-formed inputs; they either fail noisily or pass strictly clean and correct data to the next program in the chain.

However, heed also this warning:

> The original HTML documents recommended "be generous in what you accept", and it has bedeviled us ever since because each browser accepts a different

[10] Jonathan Postel was the first editor of the Internet RFC series of standards, and one of the principal architects of the Internet. A tribute page [http://www.postel.org/jonpostel.html] is maintained by the Postel Center for Experimental Networking.

> superset of the specifications. It is the *specifications* that should be generous, not their interpretation.
>
> —
> <author>DougMcIlroy</author>

McIlroy adjures us to *design* for generosity rather than compensating for inadequate standards with permissive implementations. Otherwise, as he rightly points out, it's all too easy to end up in tag soup.

Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.

In the early minicomputer days of Unix, this was still a fairly radical idea (machines were a great deal slower and more expensive then). Nowadays, with every development shop and most users (apart from the few modeling nuclear explosions or doing 3D movie animation) awash in cheap machine cycles, it may seem too obvious to need saying.

Somehow, though, practice doesn't seem to have quite caught up with reality. If we took this maxim really seriously throughout software development, most applications would be written in higher-level languages like Perl, Tcl, Python, Java, Lisp and even shell — languages that ease the programmer's burden by doing their own memory management (see [Ravenbrook]).

And indeed this is happening within the Unix world, though outside it most applications shops still seem stuck with the old-school Unix strategy of coding in C (or C++). Later in this book we'll discuss this strategy and its tradeoffs in detail.

One other obvious way to conserve programmer time is to teach machines how to do more of the low-level work of programming. This leads to...

Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.

Human beings are notoriously bad at sweating the details. Accordingly, any kind of hand-hacking of programs is a rich source of delays and errors. The simpler and more abstracted your program specification can be, the more likely it is that the human designer will have gotten it right. Generated code (at *every* level) is almost always cheaper and more reliable than hand-hacked.

We all know this is true (it's why we have compilers and interpreters, after all) but we often don't think about the implications. High-level-language code that's repetitive and mind-numbing for humans to write is just as productive a target for a code generator as machine code. It pays to use code generators when they can raise the level of abstraction — that is, when the specification language for the generator is simpler than the generated code, and the code doesn't have to be hand-hacked afterwards.

In the Unix tradition, code generators are heavily used to automate error-prone detail work. Parser/lexer generators are the classic examples; makefile generators and GUI interface builders are newer ones.

(We cover these techniques in Chapter 9.)

Rule of Optimization: Prototype before polishing. Get it working before you optimize it.

The most basic argument for prototyping first is Kernighan & Plauger's; "90% of the functionality delivered now is better than 100% of it delivered never". Prototyping first may help keep you from investing far too much time for marginal gains.

For slightly different reasons, Donald Knuth (author of *The Art Of Computer Programming*, one of the field's few true classics) popularized the observation that "Premature optimization is the root of all evil".[11] And he was right.

Rushing to optimize before the bottlenecks are known may be the only error to have ruined more designs than feature creep. From tortured code to incomprehensible data layouts, the results of obsessing about speed or memory or disk usage at the expense of transparency and simplicity are everywhere. They spawn innumerable bugs and cost millions of man-hours — often, just to get marginal gains in the use of some resource much less expensive than debugging time.

Disturbingly often, premature local optimization actually hinders global optimization (and hence reduces overall performance). A prematurely optimized portion of a design frequently interferes with changes that would have much higher payoffs across the whole design, so you end up with both inferior performance and excessively complex code.

[11] In full: "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil". Knuth himself attributes the remark to C. A. R. Hoare.

In the Unix world there is a long-established and very explicit tradition (exemplified by Rob Pike's comments above and Ken Thompson's maxim about brute force) that says: *Prototype, then polish. Get it working before you optimize it*. Or: Make it work first, then make it work fast. 'Extreme programming' guru Kent Beck, operating in a different culture, has usefully amplified this to: "Make it run, then make it right, then make it fast".

The thrust of all these quotes is the same: get your design right with an un-optimized, slow, memory-intensive implementation before you try to tune. Then, tune systematically, looking for the places where you can buy big performance wins with the smallest possible increases in local complexity.

> Prototyping is important for system design as well as optimization — it is much easier to judge whether a prototype does what you want than it is to read a long specification. I remember one development manager at Bellcore who fought against the "requirements" culture years before anybody talked about "rapid prototyping" or "agile development". He wouldn't issue long specifications; he'd lash together some combination of shell scripts and awk code that did roughly what was needed, tell the customers to send him some clerks for a few days, and then have the customers come in and look at their clerks using the prototype and tell him whether or not they liked it. If they did, he would say "you can have it industrial strength so-many-months from now at such-and-such cost". His estimates tended to be accurate, but he lost out in the culture to managers who believed that requirements writers should be in control of everything.
>
> —
>
> <author>MikeLesk</author>

Using prototyping to learn which features you don't have to implement helps optimization for performance; you don't have to optimize what you don't write. The most powerful optimization tool in existence may be the delete key.

> One of my most productive days was throwing away 1000 lines of code.
>
> —
>
> <author>KenThompson</author>

(We'll go into a bit more depth about related ideas in Chapter 12.)

Rule of Diversity: Distrust all claims for "one true way".

Even the best software tools tend to be limited by the imaginations of their designers. Nobody is smart enough to optimize for everything, nor to anticipate all the uses to which their software might be put. Designing rigid, closed software that won't talk to the rest of the world is an unhealthy form of arrogance.

Therefore, the Unix tradition includes a healthy mistrust of "one true way" approaches to software design or implementation. It embraces multiple languages, open extensible systems, and customization hooks everywhere.

Rule of Extensibility: Design for the future, because it will be here sooner than you think.

If it is unwise to trust other people's claims for "one true way", it's even more foolish to believe them about your own designs. Never assume you have the final answer. Therefore, leave room for your data formats and code to grow; otherwise, you will often find that you are locked into unwise early choices because you cannot change them while maintaining backward compatibility.

When you design protocols or file formats, make them sufficiently self-describing to be extensible. Always, *always* either include a version number, or compose the format from self-contained, self-describing clauses in such a way that new clauses can be readily added and old ones dropped without confusing format-reading code. Unix experience tells us that the marginal extra overhead of making data layouts self-describing is paid back a thousandfold by the ability to evolve them forward without breaking things.

When you design code, organize it so future developers will be able to plug new functions into the architecture without having to scrap and rebuild the architecture. This rule is not a license to add features you don't yet need; it's advice to write your code so that adding features later when you *do* need them is easy. Make the joints flexible, and put "If you ever need to..." comments in your code. You owe this grace to people who will use and maintain your code after you.

You'll be there in the future too, maintaining code you may have half forgotten under the press of more recent projects. When you design for the future, the sanity you save may be your own.

The Unix Philosophy in One Lesson

All the philosophy really boils down to one iron law, the hallowed 'KISS principle' of master engineers everywhere:



Unix gives you an excellent base for applying the KISS principle. The remainder of this book will help you learn how.

Applying the Unix Philosophy

These philosophical principles aren't just vague generalities. In the Unix world they come straight from experience and lead to specific prescriptions, some of which we've already developed above. Here's a by no means exhaustive list:

- Everything that can be a source- and destination-independent filter *should* be one.
- Data streams should if at all possible be textual (so they can be viewed and filtered with standard tools).

- Database layouts and application protocols should if at all possible be textual (human-readable and human-editable).
- Complex front ends (user interfaces) should be cleanly separated from complex back ends.
- Whenever possible, prototype in an interpreted language before coding C.
- Mixing languages is better than writing everything in one, if and only if using only that one is likely to overcomplicate the program.
- Be generous in what you accept, rigorous in what you emit.
- When filtering, never throw away information you don't need to.
- Small is beautiful. Write programs that do as little as is consistent with getting the job done.

We'll see the Unix design rules, and the prescriptions that derive from them, applied over and over again in the remainder of this book. Unsurprisingly, they tend to converge with the very best practices from software engineering in other traditions.[12]

Attitude Matters Too

When you see the right thing, do it — this may look like more work in the short term, but it's the path of least effort in the long run. If you don't know what the right thing is, do the minimum necessary to get the job done, at least until you figure out what the right thing is.

To do the Unix philosophy right, you have to be loyal to excellence. You have to believe that software design is a craft worth all the intelligence, creativity, and passion you can muster. Otherwise you won't look past the easy, stereotyped ways of approaching design and implementation; you'll rush into coding when you should be thinking. You'll carelessly complicate when you should be relentlessly simplifying — and then you'll wonder why your code bloats and debugging is so hard.

To do the Unix philosophy right, you have to value your own time enough never to waste it. If someone has already solved a problem once, don't let pride or politics suck you into solving it a

[12] One notable example is Butler Lampson's *Hints for Computer System Design* [Lampson], which I discovered late in the preparation of this book. It not only expresses a number of Unix dicta in forms that were clearly discovered independently, but uses many of the same tag lines to illustrate them.

second time rather than re-using. And never work harder than you have to; work smarter instead, and save the extra effort for when you need it. Lean on your tools and automate everything you can.

Software design and implementation should be a joyous art, a kind of high-level play. If this attitude seems preposterous or vaguely embarrassing to you, stop and think; ask yourself what you've forgotten. Why do you design software instead of doing something else to make money or pass the time? You must have thought software was worthy of your passion once....

To do the Unix philosophy right, you need to have (or recover) that attitude. You need to *care*. You need to *play*. You need to be willing to *explore*.

We hope you'll bring this attitude to the rest of this book. Or, at least, that this book will help you rediscover it.

Chapter 2. History

A Tale of Two Cultures

Those who cannot remember the past are condemned to repeat it.
--
<author>GeorgeSantayana</author>
The Life of Reason (1905)

The past informs practice. Unix has a long and colorful history, much of which is still live as folklore, assumptions, and (too often) battle scars in the collective memory of Unix programmers. In this chapter we'll survey the history of Unix, with an eye to explaining why, in 2003, today's Unix culture looks the way it does.

Origins and History of Unix, 1969-1995

A notorious ‘second-system effect‘ often afflicts the successors of small experimental prototypes. The urge to add everything that was left out the first time around all too frequently leads to huge and overcomplicated design. Less well known, because less common, is the ‘third-system effect’; sometimes, after the second system has collapsed of its own weight, there is a chance to go back to simplicity and get it really right.

The original Unix was a third system. Its grandfather was the small and simple Compatible Time-Sharing System (CTSS), either the first or second timesharing system ever deployed (depending on some definitional questions we are going to determinedly ignore). Its father was the pioneering Multics project, an attempt to create a feature-packed ‘information utility’ that would gracefully support interactive timesharing of mainframe computers by large communities of users. Multics, alas, did collapse of its own weight. But Unix was born from that collapse.

Genesis: 1969–1971

Unix was born in 1969 out of the mind of a computer scientist at Bell Laboratories, Ken Thompson. Thompson had been a researcher on the Multics project, an experience which spoiled him for the primitive batch computing that was the rule almost everywhere else. But the concept of timesharing was still a novel one in the late 1960s; the first speculations on it had been uttered barely ten years earlier by computer scientist John McCarthy (also the inventor of the Lisp language), the first actual

deployment had been in 1962, seven years earlier, and timesharing operating systems were still experimental and temperamental beasts.

Computer hardware was at that time more primitive than even people who were there to see it can now easily recall. The most powerful machines of the day had less computing power and internal memory than a typical cellphone of today.[13] Video display terminals were in their infancy and would not be widely deployed for another six years. The standard interactive device on the earliest timesharing systems was the ASR-33 teletype — a slow, noisy device that printed upper-case-only on big rolls of yellow paper. The ASR-33 was the natural parent of the Unix tradition of terse commands and sparse responses.

When Bell Labs withdrew from the Multics research consortium, Ken Thompson was left with some Multics-inspired ideas about how to build a file system. He was also left without a machine on which to play a game he had written called Space Travel, a science-fiction simulation that involved navigating a rocket through the solar system. Unix began its life on a scavenged PDP-7 minicomputer[14] like the one shown in Figure 2.1, as a platform for the Space Travel game and a testbed for Thompson's ideas about operating system design.

[13] Ken Thompson reminded me that today's cellphones have more RAM than the PDP-7 had RAM and disk storage combined; a *large* disk, in those days, was less than a megabyte of storage.

[14] There is a Web FAQ on the PDP computers [http://www.faqs.org/faqs/dec-faq/pdp8/] that explains the otherwise extremely obscure PDP-7's place in history.

Figure 2.1. The PDP-7.



The full origin story is told in [Ritchie79] from the point of view of Thompson's first collaborator, Dennis Ritchie, the man who would become known as the co-inventor of Unix and the inventor of the C language. Dennis Ritchie, Doug McIlroy, and a few colleagues had become used to interactive computing under Multics and did not want to lose that capability. Thompson's PDP-7 operating system offered them a lifeline.

Ritchie observes: "What we wanted to preserve was not just a good environment in which to do programming, but a system around which a fellowship could form. We knew from experience that the essence of communal computing, as supplied by remote-access, time-shared machines, is not just to type programs into a terminal instead of a keypunch, but to encourage close communication". The theme of computers being viewed not merely as logic devices but as the nuclei of communities was in the air; 1969 was also the year the ARPANET (the direct ancestor of today's Internet) was invented. The theme of "fellowship" would resonate all through Unix's subsequent history.

Thompson and Ritchie's Space Travel implementation attracted notice. At first, the PDP-7's software had to be cross-compiled on a GE mainframe. The utility programs that Thompson and Ritchie wrote to support hosting game development on the PDP-7 itself became the core of Unix — though the name did not attach itself until 1970. The original spelling was "UNICS" (UNiplexed

Information and Computing Service), which Ritchie later described as "a somewhat treacherous pun on Multics", which stood for MULTiplexed Information and Computing Service.

Even at its earliest stages, PDP-7 Unix bore a strong resemblance to today's Unixes and provided a rather more pleasant programming environment than was available anywhere else in those days of card-fed batch mainframes. Unix was very close to being the first system under which a programmer could sit down directly at a machine and compose programs on the fly, exploring possibilities and testing while composing. All through its lifetime Unix has had a pattern of growing more capabilities by attracting highly skilled volunteer efforts from programmers impatient with the limitations of other operating systems. This pattern was set early, within Bell Labs itself.

The Unix tradition of lightweight development and informal methods also began at its beginning. Where Multics had been a large project with thousands of pages of technical specifications written before the hardware arrived, the first running Unix code was brainstormed by three people and implemented by Ken Thompson in two days — on an obsolete machine that had been designed to be a graphics terminal for a 'real' computer.

Unix's first real job, in 1971, was to support what would now be called word processing for the Bell Labs patent department; the first Unix application was the ancestor of the nroff(1) text formatter. This project justified the purchase of a PDP-11, a much more capable minicomputer. Management remained blissfully unaware that the word-processing system that Thompson and colleagues were building was incubating an operating system. Operating systems were not in the Bell Labs plan — AT&T had joined the Multics consortium precisely to avoid doing an operating system on its own. Nevertheless, the completed system was a rousing success. It established Unix as a permanent and valued part of the computing ecology at Bell Labs, and began another theme in Unix's history — a close association with document-formatting, typesetting, and communications tools. The 1972 manual claimed 10 installations.

Later, Doug McIlroy would write of this period [McIlroy91]: "Peer pressure and simple pride in workmanship caused gobs of code to be rewritten or discarded as better or more basic ideas emerged. Professional rivalry and protection of turf were practically unknown: so many good things were happening that nobody needed to be proprietary about innovations". But it would take another quarter century for all the implications of that observation to come home.

Exodus: 1971–1980

The original Unix operating system was written in assembler, and the applications in a mix of assembler and an interpreted language called B, which had the virtue that it was small enough to

run on the PDP-7. But B was not powerful enough for systems programming, so Dennis Ritchie added data types and structures to it. The resulting C language evolved from B beginning in 1971; in 1973 Thompson and Ritchie finally succeeded in rewriting Unix in their new language. This was quite an audacious move; at the time, system programming was done in assembler in order to extract maximum performance from the hardware, and the very concept of a portable operating system was barely a gleam in anyone's eye. As late as 1979, Ritchie could write: “It seems certain that much of the success of Unix follows from the readability, modifiability, and portability of its software that in turn follows from its expression in high-level languages”, in the knowledge that this was a point that still needed making.



Ken (seated) and Dennis (standing) at a PDP-11 in 1972.

A 1974 paper in *Communications of the ACM* [Ritchie-Thompson] gave Unix its first public exposure. In that paper, its authors described the unprecedentedly simple design of Unix, and reported over 600 Unix installations. All were on machines underpowered even by the standards of that day, but (as Ritchie and Thompson wrote) “constraint has encouraged not only economy, but also a certain elegance of design”.

After the CACM paper, research labs and universities all over the world clamored for the chance to try out Unix themselves. Under a 1958 consent decree in settlement of an antitrust case, AT&T (the parent organization of Bell Labs) had been forbidden from entering the computer business. Unix could not, therefore, be turned into a product; indeed, under the terms of the consent decree, Bell Labs was required to license its nontelephone technology to anyone who asked. Ken Thompson quietly began answering requests by shipping out tapes and disk packs — each, according to legend, with a note signed “love, ken”.

This was years before personal computers. Not only was the hardware needed to run Unix too expensive to be within an individual's reach, but nobody imagined that would change in the foreseeable future. So Unix machines were only available by the grace of big organizations with big budgets: corporations, universities, government agencies. But use of these minicomputers was less regulated than the even-bigger mainframes, and Unix development rapidly took on a countercultural air. It was the early 1970s; the pioneering Unix programmers were shaggy hippies and hippie-wannabes. They delighted in playing with an operating system that not only offered them fascinating challenges at the leading edge of computer science, but also subverted all the technical

assumptions and business practices that went with Big Computing. Card punches, COBOL, business suits, and batch IBM mainframes were the despised old wave; Unix hackers reveled in the sense that they were simultaneously building the future and flipping a finger at the system.

The excitement of those days is captured in this quote from Douglas Comer: “Many universities contributed to UNIX. At the University of Toronto, the department acquired a 200-dot-per-inch printer/plotter and built software that used the printer to simulate a phototypesetter. At Yale University, students and computer scientists modified the UNIX shell. At Purdue University, the Electrical Engineering Department made major improvements in performance, producing a version of UNIX that supported a larger number of users. Purdue also developed one of the first UNIX computer networks. At the University of California at Berkeley, students developed a new shell and dozens of smaller utilities. By the late 1970s, when Bell Labs released Version 7 UNIX, it was clear that the system solved the computing problems of many departments, and that it incorporated many of the ideas that had arisen in universities. The end result was a strengthened system. A tide of ideas had started a new cycle, flowing from academia to an industrial laboratory, back to academia, and finally moving on to a growing number of commercial sites” [Comer].

The first Unix of which it can be said that essentially all of it would be recognizable to a modern Unix programmer was the Version 7 release in 1979.[15] The first Unix user group had formed the previous year. By this time Unix was in use for operations support all through the Bell System [Hauben], and had spread to universities as far away as Australia, where John Lions’s 1976 notes [Lions] on the Version 6 source code became the first serious documentation of the Unix kernel internals. Many senior Unix hackers still treasure a copy.

> The Lions book was a samizdat publishing sensation. Because of copyright infringement or some such it couldn’t be published in the U.S., so copies of copies seeped everywhere. I still have my copy, which was at least 6th generation. Back then you couldn’t be a kernel hacker without a Lions.
>
> —
>
> <author>KenArnold</author>

The beginnings of a Unix industry were coalescing as well. The first Unix company (the Santa Cruz Operation, SCO) began operations in 1978, and the first commercial C compiler (Whitesmiths) sold that same year. By 1980 an obscure software company in Seattle was also getting into the Unix game, shipping a port of the AT&T version for microcomputers called XENIX. But

[15] The version 7 manuals can be browsed on-line at http://plan9.bell-labs.com/7thEdMan/index.html.

Microsoft's affection for Unix as a product was not to last very long (though Unix would continue to be used for most internal development work at the company until after 1990).

TCP/IP and the Unix Wars: 1980-1990

The Berkeley campus of the University of California emerged early as the single most important academic hot-spot in Unix development. Unix research had begun there in 1974, and was given a substantial impetus when Ken Thompson taught at the University during a 1975-76 sabbatical. The first BSD release had been in 1977 from a lab run by a then-unknown grad student named Bill Joy. By 1980 Berkeley was the hub of a sub-network of universities actively contributing to their variant of Unix. Ideas and code from Berkeley Unix (including the vi(1) editor) were feeding back from Berkeley to Bell Labs.

Then, in 1980, the Defense Advanced Research Projects Agency needed a team to implement its brand-new TCP/IP protocol stack on the VAX under Unix. The PDP-10s that powered the ARPANET at that time were aging, and indications that DEC might be forced to cancel the 10 in order to support the VAX were already in the air. DARPA considered contracting DEC to implement TCP/IP, but rejected that idea because they were concerned that DEC might not be responsive to requests for changes in their proprietary VAX/VMS operating system [Libes-Ressler]. Instead, DARPA chose Berkeley Unix as a platform — explicitly because its source code was available and unencumbered [Leonard].

Berkeley's Computer Science Research Group was in the right place at the right time with the strongest development tools; the result became arguably the most critical turning point in Unix's history since its invention.

Until the TCP/IP implementation was released with Berkeley 4.2 in 1983, Unix had had only the weakest networking support. Early experiments with Ethernet were unsatisfactory. An ugly but serviceable facility called UUCP (Unix to Unix Copy Program) had been developed at Bell Labs for distributing software over conventional telephone lines via modem.[16] UUCP could forward Unix mail between widely separated machines, and (after Usenet was invented in 1981) supported Usenet, a distributed bulletin-board facility that allowed users to broadcast text messages to anywhere that had phone lines and Unix systems.

Still, the few Unix users aware of the bright lights of the ARPANET felt like they were stuck in a backwater. No FTP, no telnet, only the most restricted remote job execution, and painfully

[16] UUCP was hot stuff when a *fast* modem was 300 baud.

slow links. Before TCP/IP, the Internet and Unix cultures did not mix. Dennis Ritchie's vision of computers as a way to "encourage close communication" was one of collegial communities clustered around individual timesharing machines or in the same computing center; it didn't extend to the continent-wide distributed 'network nation' that ARPA users had started to form in the mid-1970s. Early ARPANETters, for their part, considered Unix a crude makeshift limping along on risibly weak hardware.

After TCP/IP, everything changed. The ARPANET and Unix cultures began to merge at the edges, a development that would eventually save both from destruction. But there would be hell to pay first as the result of two unrelated disasters; the rise of Microsoft and the AT&T divestiture.

In 1981, Microsoft made its historic deal with IBM over the new IBM PC. Bill Gates bought QDOS (Quick and Dirty Operating System), a clone of CP/M that its programmer Tim Paterson had thrown together in six weeks, from Paterson's employer Seattle Computer Products. Gates, concealing the IBM deal from Paterson and SCP, bought the rights for $50,000. He then talked IBM into allowing Microsoft to market MS-DOS separately from the PC hardware. Over the next decade, leveraging code he didn't write made Bill Gates a multibillionaire, and business tactics even sharper than the original deal gained Microsoft a monopoly lock on desktop computing. XENIX as a product was rapidly deep-sixed, and eventually sold to SCO.

It was not apparent at the time how successful (or how destructive) Microsoft was going to be. Since the IBM PC-1 didn't have the hardware capacity to run Unix, Unix people barely noticed it at all (though, ironically enough, DOS 2.0 eclipsed CP/M largely because Microsoft's co-founder Paul Allen merged in Unix features including subdirectories and pipes). There were things that seemed much more interesting going on — like the 1982 launching of Sun Microsystems.

Sun Microsystems founders Bill Joy, Andreas Bechtolsheim, and Vinod Khosla set out to build a dream Unix machine with built-in networking capability. They combined hardware designed at Stanford with the Unix developed at Berkeley to produce a smashing success, and founded the workstation industry. At the time, nobody much minded watching source-code access to one branch of the Unix tree gradually dry up as Sun began to behave less like a freewheeling startup and more like a conventional firm. Berkeley was still distributing BSD with source code. Officially, System III source licenses cost $40,000 each; but Bell Labs was turning a blind eye to the number of bootleg Bell Labs Unix tapes in circulation, the universities were still swapping code with Bell Labs, and it looked like Sun's commercialization of Unix might just be the best thing to happen to it yet.

1982 was also the year that C first showed signs of establishing itself outside the Unix world as the systems-programming language of choice. It would only take about five years for C to drive

machine assemblers almost completely out of use. By the early 1990s C and C++ would dominate not only systems but application programming; by the late 1990s all other conventional compiled languages would be effectively obsolete.

When DEC canceled development on the PDP-10's successor machine (Jupiter) in 1983, VAXes running Unix began to take over as the dominant Internet machines, a position they would hold until being displaced by Sun workstations. By 1985, about 25% of all VAXes would be running Unix despite DEC's stiff opposition. But the longest-term effect of the Jupiter cancellation was a less obvious one; the death of the MIT AI Lab's PDP-10-centered hacker culture motivated a programmer named Richard Stallman to begin writing GNU, a complete free clone of Unix.

By 1983 there were no fewer than six Unix-workalike operating systems for the IBM-PC: uNETix, Venix, Coherent, QNX, Idris, and the port hosted on the Sritek PC daughtercard. There was still no port of Unix in either the System V or BSD versions; both groups considered the 8086 microprocessor woefully underpowered and wouldn't go near it. None of the Unix-workalikes were significant as commercial successes, but they indicated a significant demand for Unix on cheap hardware that the major vendors were not supplying. No individual could afford to meet it, either, not with the $40,000 price-tag on a source-code license.

Sun was already a success (with imitators!) when, in 1983, the U.S. Department of Justice won its second antitrust case against AT&T and broke up the Bell System. This relieved AT&T from the 1958 consent decree that had prevented them from turning Unix into a product. AT&T promptly rushed to commercialize Unix System V—a move that nearly killed Unix.

> So true. But their marketing did spread Unix internationally.
>
> —
>
> <author>KenThompson</author>

Most Unix boosters thought that the divestiture was great news. We thought we saw in the post-divestiture AT&T, Sun Microsystems, and Sun's smaller imitators the nucleus of a healthy Unix industry — one that, using inexpensive 68000-based workstations, would challenge and eventually break the oppressive monopoly that then loomed over the computer industry — IBM's.

What none of us realized at the time was that the productization of Unix would destroy the free exchanges of source code that had nurtured so much of the system's early vitality. Knowing no other model than secrecy for collecting profits from software and no other model than centralized control for developing a commercial product, AT&T clamped down hard on source-code distribution.

Bootleg Unix tapes became far less interesting in the knowledge that the threat of lawsuit might come with them. Contributions from universities began to dry up.

To make matters worse, the big new players in the Unix market promptly committed major strategic blunders. One was to seek advantage by product differentiation — a tactic which resulted in the interfaces of different Unixes diverging. This threw away cross-platform compatibility and fragmented the Unix market.

The other, subtler error was to behave as if personal computers and Microsoft were irrelevant to Unix's prospects. Sun Microsystems failed to see that commoditized PCs would inevitably become an attack on its workstation market from below. AT&T, fixated on minicomputers and mainframes, tried several different strategies to become a major player in computers, and badly botched all of them. A dozen small companies formed to support Unix on PCs; all were underfunded, focused on selling to developers and engineers, and never aimed at the business and home market that Microsoft was targeting.

In fact, for years after divestiture the Unix community was preoccupied with the first phase of the Unix wars — an internal dispute, the rivalry between System V Unix and BSD Unix. The dispute had several levels, some technical (sockets vs. streams, BSD tty vs. System V termio) and some cultural. The divide was roughly between longhairs and shorthairs; programmers and technical people tended to line up with Berkeley and BSD, more business-oriented types with AT&T and System V. The longhairs, repeating a theme from Unix's early days ten years before, liked to see themselves as rebels against a corporate empire; one of the small companies put out a poster showing an X-wing-like space fighter marked "BSD" speeding away from a huge AT&T 'death star' logo left broken and in flames. Thus we fiddled while Rome burned.

But something else happened in the year of the AT&T divestiture that would have more long-term importance for Unix. A programmer/linguist named Larry Wall quietly invented the patch(1) utility. The *patch* program, a simple tool that applies changebars generated by diff(1) to a base file, meant that Unix developers could cooperate by passing around patch sets — incremental changes to code — rather than entire code files. This was important not only because patches are less bulky than full files, but because patches would often apply cleanly even if much of the base file had changed since the patch-sender fetched his copy. With this tool, streams of development on a common source-code base could diverge, run in parallel, and re-converge. The *patch* program did more than any other single tool to enable collaborative development over the Internet — a method that would revitalize Unix after 1990.

In 1985 Intel shipped the first 386 chip, capable of addressing 4 gigabytes of memory with a flat address space. The clumsy segment addressing of the 8086 and 286 became immediately obsolete. This was big news, because it meant that for the first time, a microprocessor in the dominant Intel family had the capability to run Unix without painful compromises. The handwriting was on the wall for Sun and the other workstation makers. They failed to see it.

1985 was also the year that Richard Stallman issued the GNU manifesto [Stallman] and launched the Free Software Foundation. Very few people took him or his GNU project seriously, a judgment that turned out to be seriously mistaken. In an unrelated development of the same year, the originators of the X window system released it as source code without royalties, restrictions, or license code. As a direct result of this decision, it became a safe neutral area for collaboration between Unix vendors, and defeated proprietary contenders to become Unix's graphics engine.

Serious standardization efforts aimed at reconciling the System V and Berkeley APIs also began in 1983 with the /usr/group standard. This was followed in 1985 by the POSIX standards, an effort backed by the IEEE. These described the intersection set of the BSD and SVR3 (System V Release 3) calls, with the superior Berkeley signal handling and job control but with SVR3 terminal control. All later Unix standards would incorporate POSIX at their core, and later Unixes would adhere to it closely. The only major addition to the modern Unix kernel API to come afterwards was BSD sockets.

In 1986 Larry Wall, previously the inventor of patch(1), began work on Perl, which would become the first and most widely used of the open-source scripting languages. In early 1987 the first version of the GNU C compiler appeared, and by the end of 1987 the core of the GNU toolset was falling into place: editor, compiler, debugger, and other basic development tools. Meanwhile, the X windowing system was beginning to show up on relatively inexpensive workstations. Together, these would provide the armature for the open-source Unix developments of the 1990s.

1986 was also the year that PC technology broke free of IBM's grip. IBM, still trying to preserve a price-vs.-power curve across its product line that would favor its high-margin mainframe business, rejected the 386 for most of its new line of PS/2 computers in favor of the weaker 286. The PS/2 series, designed around a proprietary bus architecture to lock out clonemakers, became a colossally expensive failure.[17] Compaq, the most aggressive of the clonemakers, trumped IBM's move by releasing the first 386 machine. Even with a clock speed of a mere 16 MHz, the 386 made a tolerable Unix machine. It was the first PC of which that could be said.

[17] The PS/2 did, however, leave one mark on later PCs — they made the mouse a standard peripheral, which is why the mouse connector on the back of your chassis is called a "PS/2 port".

It was beginning to be possible to imagine that Stallman's GNU project might mate with 386 machines to produce Unix workstations almost an order of magnitude less costly than anyone was offering. Curiously, no one seems to have actually got this far in their thinking. Most Unix programmers, coming from the minicomputer and workstation worlds, continued to disdain cheap 80x86 machines in favor of more elegant 68000-based designs. And, though a lot of programmers contributed to the GNU project, among Unix people it tended to be considered a quixotic gesture that was unlikely to have near-term practical consequences.

The Unix community had never lost its rebel streak. But in retrospect, we were nearly as blind to the future bearing down on us as IBM or AT&T. Not even Richard Stallman, who had declared a moral crusade against proprietary software a few years before, really understood how badly the productization of Unix had damaged the community around it; his concerns were with more abstract and long-term issues. The rest of us kept hoping that some clever variation on the corporate formula would solve the problems of fragmentation, wretched marketing, and strategic drift, and redeem Unix's pre-divestiture promise. But worse was still to come.

1988 was the year Ken Olsen (CEO of DEC) famously described Unix as “snake oil”. DEC had been shipping its own variant of Unix on PDP-11s since 1982, but really wanted the business to go to its proprietary VMS operating system. DEC and the minicomputer industry were in deep trouble, swamped by waves of powerful low-cost machines coming out of Sun Microsystems and the rest of the workstation vendors. Most of those workstations ran Unix.

But the Unix industry's own problems were growing more severe. In 1988 AT&T took a 20% stake in Sun Microsystems. These two companies, the leaders in the Unix market, were beginning to wake up to the threat posed by PCs, IBM, and Microsoft, and to realize that the preceding five years of bloodletting had gained them little. The AT&T/Sun alliance and the development of technical standards around POSIX eventually healed the breach between the System V and BSD Unix lines. But the second phase of the Unix wars began when the second-tier vendors (IBM, DEC, Hewlett-Packard, and others) formed the Open Software Foundation and lined up against the AT&T/Sun axis (represented by Unix International). More rounds of Unix fighting Unix ensued.

Meanwhile, Microsoft was making billions in the home and small-business markets that the warring Unix factions had never found the will to address. The 1990 release of Windows 3.0 — the first successful graphical operating system from Redmond — cemented Microsoft's dominance, and created the conditions that would allow them to flatten and monopolize the market for desktop applications in the 1990s.

The years from 1989 to 1993 were the darkest in Unix's history. It appeared then that all the Unix community's dreams had failed. Internecine warfare had reduced the proprietary Unix industry to a squabbling shambles that never summoned either the determination or the capability to challenge Microsoft. The elegant Motorola chips favored by most Unix programmers had lost out to Intel's ugly but inexpensive processors. The GNU project failed to produce the free Unix kernel it had been promising since 1985, and after years of excuses its credibility was beginning to wear thin. PC technology was being relentlessly corporatized. The pioneering Unix hackers of the 1970s were hitting middle age and slowing down. Hardware was getting cheaper, but Unix was still too expensive. We were belatedly becoming aware that the old monopoly of IBM had yielded to a newer monopoly of Microsoft, and Microsoft's mal-engineered software was rising around us like a tide of sewage.

Blows against the Empire: 1991-1995

The first glimmer of light in the darkness was the 1990 effort by William Jolitz to port BSD onto a 386 box, publicized by a series of magazine articles beginning in 1991. The 386BSD port was possible because, partly influenced by Stallman, Berkeley hacker Keith Bostic had begun an effort to clean AT&T proprietary code out of the BSD sources in 1988. But the 386BSD project took a severe blow when, near the end of 1991, Jolitz walked away from it and destroyed his own work. There are conflicting explanations, but a common thread in all is that Jolitz wanted his code to be released as unencumbered source and was upset when the corporate sponsors of the project opted for a more proprietary licensing model.

In August 1991 Linus Torvalds, then an unknown university student from Finland, announced the Linux project. Torvalds is on record that one of his main motivations was the high cost of Sun's Unix at his university. Torvalds has also said that he would have joined the BSD effort had he known of it, rather than founding his own. But 386BSD was not shipped until early 1992, some months after the first Linux release.

The importance of both these projects became clear only in retrospect. At the time, they attracted little notice even within the Internet hacker culture — let alone in the wider Unix community, which was still fixated on more capable machines than PCs, and on trying to reconcile the special properties of Unix with the conventional proprietary model of a software business.

It would take another two years and the great Internet explosion of 1993–1994 before the true importance of Linux and the open-source BSD distributions became evident to the rest of the Unix world. Unfortunately for the BSDers, an AT&T lawsuit against BSDI (the startup company that had

backed the Jolitz port) consumed much of that time and motivated some key Berkeley developers to switch to Linux.

> Code copying and theft of trade secrets was alleged. The actual infringing code was not identified for nearly two years. The lawsuit could have dragged on for much longer but for the fact that Novell bought USL from AT&T and sought a settlement. In the end, three files were removed from the 18,000 that made up the distribution, and a number of minor changes were made to other files. In addition, the University agreed to add USL copyrights to about 70 files, with the stipulation that those files continued to be freely redistributed.
>
> —
> <author>MarshallKirkMcKusick</author>

The settlement set an important precedent by freeing an entire working Unix from proprietary control, but its effects on BSD itself were dire. Matters were not helped when, in 1992–1994, the Computer Science Research Group at Berkeley shut down; afterwards, factional warfare within the BSD community split it into three competing development efforts. As a result, the BSD lineage lagged behind Linux at a crucial time and lost to it the lead position in the Unix community.

The Linux and BSD development efforts were native to the Internet in a way previous Unixes had not been. They relied on distributed development and Larry Wall's patch(1) tool, and recruited developers via email and through Usenet newsgroups. Accordingly, they got a tremendous boost when Internet Service Provider businesses began to proliferate in 1993, enabled by changes in telecomm technology and the privatization of the Internet backbone that are outside the scope of this history. The demand for cheap Internet was created by something else: the 1991 invention of the World Wide Web. The Web was the "killer app" of the Internet, the graphical user interface technology that made it irresistible to a huge population of nontechnical end users.

The mass-marketing of the Internet both increased the pool of potential developers and lowered the transaction costs of distributed development. The results were reflected in efforts like XFree86, which used the Internet-centric model to build a more effective development organization than that of the official X Consortium. The first XFree86 in 1992 gave Linux and the BSDs the graphical-user-interface engine they had been missing. Over the next decade XFree86 would lead in X development, and an increasing portion of the X Consortium's activity would come to consist of funneling innovations originated in the XFree86 community back to the Consortium's industrial sponsors.

By late 1993, Linux had both Internet capability and X. The entire GNU toolkit had been hosted on it from the beginning, providing high-quality development tools. Beyond GNU tools, Linux acted as a basin of attraction, collecting and concentrating twenty years of open-source software that had previously been scattered across a dozen different proprietary Unix platforms. Though the Linux kernel was still officially in beta (at 0.99 level), it was remarkably crash-free. The breadth and quality of the software in Linux distributions was already that of a production-ready operating system.

A few of the more flexible-minded among old-school Unix developers began to notice that the long-awaited dream of a cheap Unix system for everybody had snuck up on them from an unexpected direction. It didn't come from AT&T or Sun or any of the traditional vendors. Nor did it rise out of an organized effort in academia. It was a bricolage that bubbled up out of the Internet by what seemed like spontaneous generation, appropriating and recombining elements of the Unix tradition in surprising ways.

Elsewhere, corporate maneuvering continued. AT&T divested its interest in Sun in 1992; then sold its Unix Systems Laboratories to Novell in 1993; Novell handed off the Unix trademark to the X/Open standards group in 1994; AT&T and Novell joined OSF in 1994, finally ending the Unix wars. In 1995 SCO bought UnixWare (and the rights to the original Unix sources) from Novell. In 1996, X/Open and OSF merged, creating one big Unix standards group.

But the conventional Unix vendors and the wreckage of their wars came to seem steadily less and less relevant. The action and energy in the Unix community were shifting to Linux and BSD and the open-source developers. By the time IBM, Intel, and SCO announced the Monterey project in 1998 — a last-gasp attempt to merge One Big System out of all the proprietary Unixes left standing — developers and the trade press reacted with amusement, and the project was abruptly canceled in 2001 after three years of going nowhere.

The industry transition could not be said to have completed until 2000, when SCO sold UnixWare and the original Unix source-code base to Caldera — a Linux distributor. But after 1995, the story of Unix became the story of the open-source movement. There's another side to that story; to tell it, we'll need to return to 1961 and the origins of the Internet hacker culture.

Origins and History of the Hackers, 1961-1995

The Unix tradition is an implicit culture that has always carried with it more than just a bag of technical tricks. It transmits a set of values about beauty and good design; it has legends and folk heroes. Intertwined with the history of the Unix tradition is another implicit culture that is more

difficult to label neatly. It has its own values and legends and folk heroes, partly overlapping with those of the Unix tradition and partly derived from other sources. It has most often been called the “hacker culture”, and since 1998 has largely coincided with what the computer trade press calls “the open source movement”.

The relationships between the Unix tradition, the hacker culture, and the open-source movement are subtle and complex. They are not simplified by the fact that all three implicit cultures have frequently been expressed in the behaviors of the same human beings. But since 1990 the story of Unix is largely the story of how the open-source hackers changed the rules and seized the initiative from the old-line proprietary Unix vendors. Therefore, the other half of the history behind today’s Unix is the history of the hackers.

At Play in the Groves of Academe: 1961-1980

The roots of the hacker culture can be traced back to 1961, the year MIT took delivery of its first PDP-1 minicomputer. The PDP-1 was one of the earliest interactive computers, and (unlike other machines) of the day was inexpensive enough that time on it did not have to be rigidly scheduled. It attracted a group of curious students from the Tech Model Railroad Club who experimented with it in a spirit of fun. *Hackers: Heroes of the Computer Revolution* [Levy] entertainingly describes the early days of the club. Their most famous achievement was SPACEWAR, a game of dueling rocketships loosely inspired by the *Lensman* space operas of E.E. “Doc” Smith.[18]

Several of the TMRC experimenters later went on to become core members of the MIT Artificial Intelligence Lab, which in the 1960s and 1970s became one of the world centers of cutting-edge computer science. They took some of TMRC’s slang and in-jokes with them, including a tradition of elaborate (but harmless) pranks called “hacks”. The AI Lab programmers appear to have been the first to describe themselves as “hackers”.

After 1969 the MIT AI Lab was connected, via the early ARPANET, to other leading computer science research laboratories at Stanford, Bolt Beranek & Newman, Carnegie-Mellon University and elsewhere. Researchers and students got the first foretaste of the way fast network access abolishes geography, often making it easier to collaborate and form friendships with distant people on the net than it would be to do likewise with colleagues closer-by but less connected.

Software, ideas, slang, and a good deal of humor flowed over the experimental ARPANET links. Something like a shared culture began to form. One of its earliest and most enduring artifacts was

[18]SPACEWAR was not related to Ken Thompson’s Space Travel game, other than by the fact that both appealed to science-fiction fans.

the Jargon File, a list of shared slang terms that originated at Stanford in 1973 and went through several revisions at MIT after 1976. Along the way it accumulated slang from CMU, Yale, and other ARPANET sites.

Technically, the early hacker culture was largely hosted on PDP-10 minicomputers. They used a variety of operating systems that have since passed into history: TOPS-10, TOPS-20, Multics, ITS, SAIL. They programmed in assembler and dialects of Lisp. PDP-10 hackers took over running the ARPANET itself because nobody else wanted the job. Later, they became the founding cadre of the Internet Engineering Task Force (IETF) and originated the tradition of standardization through Requests For Comment (RFCs).

Socially, they were young, exceptionally bright, almost entirely male, dedicated to programming to the point of addiction, and tended to have streaks of stubborn nonconformism — what years later would be called 'geeks'. They, too, tended to be shaggy hippies and hippie-wannabes. They, too, had a vision of computers as community-building devices. They read Robert Heinlein and J. R. R. Tolkien, played in the Society for Creative Anachronism, and tended to have a weakness for puns. Despite their quirks (or perhaps because of them!) many of them were among the brightest programmers in the world.

They were *not* Unix programmers. The early Unix community was drawn largely from the same pool of geeks in academia and government or commercial research laboratories, but the two cultures differed in important ways. One that we've already touched on is the weak networking of early Unix. There was effectively no Unix-based ARPANET access until after 1980, and it was uncommon for any individual to have a foot in both camps.

Collaborative development and the sharing of source code was a valued tactic for Unix programmers. To the early ARPANET hackers, on the other hand, it was more than a tactic: it was something rather closer to a shared religion, partly arising from the academic "publish or perish" imperative and (in its more extreme versions) developing into an almost Chardinist idealism about networked communities of minds. The most famous of these hackers, Richard M. Stallman, became the ascetic saint of that religion.

Internet Fusion and the Free Software Movement: 1981-1991

After 1983 and the BSD port of TCP/IP, the Unix and ARPANET cultures began to fuse together. This was a natural development once the communication links were in place, since both cultures were composed of the same kind of people (indeed, in a few but significant cases the *same* people). ARPANET hackers learned C and began to speak the jargon of pipes, filters, and shells; Unix

programmers learned TCP/IP and started to call each other “hackers”. The process of fusion was accelerated after the Project Jupiter cancellation in 1983 killed the PDP-10’s future. By 1987 the two cultures had merged so completely that most hackers programmed in C and casually used slang terms that went back to the Tech Model Railroad Club of twenty-five years earlier.

(In 1979 I was unusual in having strong ties to both the Unix and ARPANET cultures. In 1985 that was no longer unusual. By the time I expanded the old ARPANET Jargon File into the *New Hacker’s Dictionary* [Raymond96] in 1991, the two cultures had effectively fused. The Jargon File, born on the ARPANET but revised on Usenet, aptly symbolized the merger.)

But TCP/IP networking and slang were not the only things the post-1980 hacker culture inherited from its ARPANET roots. It also got Richard Stallman, and Stallman’s moral crusade.

Richard M. Stallman (generally known by his login name, RMS) had already proved by the late 1970s that he was one of the most able programmers alive. Among his many inventions was the Emacs editor. For RMS, the Jupiter cancellation in 1983 only finished off a disintegration of the MIT AI Lab culture that had begun a few years earlier as many of its best went off to help run competing Lisp-machine companies. RMS felt ejected from a hacker Eden, and decided that proprietary software was to blame.

In 1983 Stallman founded the GNU project, aimed at writing an entire free operating system. Though Stallman was not and had never been a Unix programmer, under post-1980 conditions implementing a Unix-like operating system became the obvious strategy to pursue. Most of RMS’s early contributors were old-time ARPANET hackers newly decanted into Unix-land, in whom the ethos of code-sharing ran rather stronger than it did among those with a more Unix-centered background.

In 1985, RMS published the GNU Manifesto. In it he consciously created an ideology out of the values of the pre-1980 ARPANET hackers — complete with a novel ethico-political claim, a self-contained and characteristic discourse, and an activist plan for change. RMS aimed to knit the diffuse post-1980 community of hackers into a coherent social machine for achieving a single revolutionary purpose. His behavior and rhetoric half-consciously echoed Karl Marx’s attempts to mobilize the industrial proletariat against the alienation of their work.

RMS’s manifesto ignited a debate that is still live in the hacker culture today. His program went way beyond maintaining a codebase, and essentially implied the abolition of intellectual-property rights in software. In pursuit of this goal, RMS popularized the term “free software”, which was the first attempt to label the product of the entire hacker culture. He wrote the General Public License

(GPL), which was to become both a rallying point and a focus of great controversy, for reasons we will examine in Chapter 16. You can learn more about RMS's position and the Free Software Foundation at the GNU website [http://www.gnu.org].

The term "free software" was partly a description and partly an attempt to define a cultural identity for hackers. On one level, it was quite successful. Before RMS, people in the hacker culture recognized each other as fellow-travelers and used the same slang, but nobody bothered arguing about what a 'hacker' is or should be. After him, the hacker culture became much more self-conscious; value disputes (often framed in RMS's language even by those who opposed his conclusions) became a normal feature of debate. RMS, a charismatic and polarizing figure, himself became so much a culture hero that by the year 2000 he could hardly be distinguished from his legend. *Free as in Freedom* [Williams] gives us an excellent portrait.

RMS's arguments influenced the behavior even of many hackers who remained skeptical of his theories. In 1987, he persuaded the caretakers of BSD Unix that cleaning out AT&T's proprietary code so they could release an unencumbered version would be a good idea. However, despite his determined efforts over more than fifteen years, the post-1980 hacker culture never unified around his ideological vision.

Other hackers were rediscovering open, collaborative development without secrets for more pragmatic, less ideological reasons. A few buildings away from Richard Stallman's 9th-floor office at MIT, the X development team thrived during the late 1980s. It was funded by Unix vendors who had argued each other to a draw over the control and intellectual-property-rights issues surrounding the X windowing system, and saw no better alternative than to leave it free to everyone. In 1987–1988 the X development prefigured the really huge distributed communities that would redefine the leading edge of Unix five years later.

> X was one of the first large-scale open-source projects to be developed by a disparate team of individuals working for different organizations spread across the globe. E-mail allowed ideas to move rapidly among the group so that issues could be resolved as quickly as necessary, and each individual could contribute in whatever capacity suited them best. Software updates could be distributed in a matter of hours, enabling every site to act in a concerted manner during development. The net changed the way software could be developed.
>
> —
> <author>KeithPackard</author>

The X developers were no partisans of the GNU master plan, but they weren't actively opposed to it, either. Before 1995 the most serious opposition to the GNU plan came from the BSD developers. The BSD people, who remembered that they had been writing freely redistributable and modifiable software years before RMS's manifesto, rejected GNU's claim to historical and ideological primacy. They specifically objected to the infectious or "viral" property of the GPL, holding out the BSD license as being "more free" because it placed fewer restrictions on the reuse of code.

It did not help RMS's case that, although his Free Software Foundation had produced most of the rest of a full software toolkit, it failed to deliver the central piece. Ten years after the founding of the GNU project, there was still no GNU kernel. While individual tools like Emacs and GCC proved tremendously useful, GNU without a kernel neither threatened the hegemony of proprietary Unixes nor offered an effective counter to the rising problem of the Microsoft monopoly.

After 1995 the debate over RMS's ideology took a somewhat different turn. Opposition to it became closely associated with both Linus Torvalds and the author of this book.

Linux and the Pragmatist Reaction: 1991-1998

Even as the HURD (the GNU kernel) effort was stalling, new possibilities were opening up. In the early 1990s the combination of cheap, powerful PCs with easy Internet access proved a powerful lure for a new generation of young programmers looking for challenges to test their mettle. The user-space toolkit written by the Free Software Foundation suggested a way forward that was free of the high cost of proprietary software development tools. Ideology followed economics rather than leading the charge; some of the newbies signed up with RMS's crusade and adopted the GPL as their banner, and others identified more with the Unix tradition as a whole and joined the anti-GPL camp, but most dismissed the whole dispute as a distraction and just wrote code.

Linus Torvalds neatly straddled the GPL/anti-GPL divide by using the GNU toolkit to surround the Linux kernel he had invented and the GPL's infectious properties to protect it, but rejecting the ideological program that went with RMS's license. Torvalds affirmed that he thought free software better in general but occasionally used proprietary programs. His refusal to be a zealot even in his own cause made him tremendously attractive to the majority of hackers who had been uncomfortable with RMS's rhetoric, but had lacked any focus or convincing spokesperson for their skepticism.

Torvalds's cheerful pragmatism and adept but low-key style catalyzed an astonishing string of victories for the hacker culture in the years 1993–1997, including not merely technical successes but the solid beginnings of a distribution, service, and support industry around the Linux operating system. As a result his prestige and influence skyrocketed. Torvalds became a hero on Internet

time; by 1995, he had achieved in just four years the kind of culture-wide eminence that RMS had required fifteen years to earn — and far exceeded Stallman's record at selling "free software" to the outside world. By contrast with Torvalds, RMS's rhetoric began to seem both strident and unsuccessful.

Between 1991 and 1995 Linux went from a proof-of-concept surrounding an 0.1 prototype kernel to an operating system that could compete on features and performance with proprietary Unixes, and beat most of them on important statistics like continuous uptime. In 1995, Linux found its killer app: Apache, the open-source webserver. Like Linux, Apache proved remarkably stable and efficient. Linux machines running Apache quickly became the platform of choice for ISPs worldwide; Apache captured about 60% of websites,[19] handily beating out both of its major proprietary competitors.

The one thing Torvalds did not offer was a new ideology — a new rationale or generative myth of hacking, and a positive discourse to replace RMS's hostility to intellectual property with a program more attractive to people both within and outside the hacker culture. I inadvertently supplied this lack in 1997 as a result of trying to understand why Linux's development had not collapsed in confusion years before. The technical conclusions of my published papers [Raymond01] will be summarized in Chapter 19. For this historical sketch, it will be sufficient to note the impact of the first one's central formula: "Given a sufficiently large number of eyeballs, all bugs are shallow".

This observation implied something nobody in the hacker culture had dared to really believe in the preceding quarter-century: that its methods could reliably produce software that was not just more elegant but more reliable and *better* than our proprietary competitors' code. This consequence, quite unexpectedly, turned out to present exactly the direct challenge to the discourse of "free software" that Torvalds himself had never been interested in mounting. For most hackers and almost all nonhackers, "Free software because it works better" easily trumped "Free software because all software should be free".

The paper's contrast between 'cathedral' (centralized, closed, controlled, secretive) and 'bazaar' (decentralized, open, peer-review-intensive) modes of development became a central metaphor in the new thinking. In an important sense this was merely a return to Unix's pre-divestiture roots — it is continuous with McIlroy's 1991 observations about the positive effects of peer pressure on Unix development in the early 1970s and Dennis Ritchie's 1979 reflections on fellowship, cross-fertilized with the early ARPANET's academic tradition of peer review and with its idealism about distributed communities of mind.

[19] Current and historical webserver share figures are available at the monthly Netcraft Web Server Survey [http://www.netcraft.com/survey/].

In early 1998, the new thinking helped motivate Netscape Communications to release the source code of its Mozilla browser. The press attention surrounding that event took Linux to Wall Street, helped drive the technology-stock boom of 1999–2001, and proved to be a turning point in both the history of the hacker culture and of Unix.

The Open-Source Movement: 1998 and Onward

By the time of the Mozilla release in 1998, the hacker community could best be analyzed as a loose collection of factions or tribes that included Richard Stallman's Free Software Movement, the Linux community, the Perl community, the Apache community, the BSD community, the X developers, the Internet Engineering Task Force (IETF), and at least a dozen others. These factions overlap, and an individual developer would be quite likely to be affiliated with two or more.

A tribe might be grouped around a particular codebase that they maintain, or around one or more charismatic influence leaders, or around a language or development tool, or around a particular software license, or around a technical standard, or around a caretaker organization for some part of the infrastructure. Prestige tends to correlate with longevity and historical contribution as well as more obvious drivers like current market-share and mind-share; thus, perhaps the most universally respected of the tribes is the IETF, which can claim continuity back to the beginnings of the ARPANET in 1969. The BSD community, with continuous traditions back to the late 1970s, commands considerable prestige despite having a much lower installation count than Linux. Stallman's Free Software Movement, dating back to the early 1980s, ranks among the senior tribes both on historical contribution and as the maintainer of several of the software tools in heaviest day-to-day use.

After 1995 Linux acquired a special role as both the unifying platform for most of the community's other software and the hackers' most publicly recognizable brand name. The Linux community showed a corresponding tendency to absorb other sub-tribes — and, for that matter, to co-opt and absorb the hacker factions associated with proprietary Unixes. The hacker culture as a whole began to draw together around a common mission: push Linux and the bazaar development model as far as it could go.

Because the post-1980 hacker culture had become so deeply rooted in Unix, the new mission was implicitly a brief for the triumph of the Unix tradition. Many of the hacker community's senior leaders were also Unix old-timers, still bearing scars from the post-divestiture civil wars of the 1980s and getting behind Linux as the last, best hope to fulfill the rebel dreams of the early Unix days.

The Mozilla release helped further concentrate opinions. In March of 1998 an unprecedented summit meeting of community influence leaders representing almost all of the major tribes convened to consider common goals and tactics. That meeting adopted a new label for the common development method of all the factions: open source.

Within six months almost all the tribes in the hacker community would accept “open source” as its new banner. Older groups like IETF and the BSD developers would begin to apply it retrospectively to what they had been doing all along. In fact, by 2000 the rhetoric of open source would not just unify the hacker culture’s present practice and plans for the future, but re-color its view of its own past.

The galvanizing effect of the Netscape announcement, and of the new prominence of Linux, reached well beyond the Unix community and the hacker culture. Beginning in 1995, developers from various platforms in the path of Microsoft’s Windows juggernaut (MacOS; Amiga; OS/2; DOS; CP/M; the weaker proprietary Unixes; various mainframe, minicomputer, and obsolete microcomputer operating systems) had banded together around Sun Microsystems’s Java language. Many disgruntled Windows developers joined them in hopes of maintaining at least some nominal independence from Microsoft. But Sun’s handling of Java was (as we discuss in Chapter 14) clumsy and alienating on several levels. Many Java developers liked what they saw in the nascent open-source movement, and followed Netscape’s lead into Linux and open source just as they had previously followed Netscape into Java.

Open-source activists welcomed the surge of immigrants from everywhere. The old Unix hands began to share the new immigrants’ dreams of not merely passively out-enduring the Microsoft monopoly, but actually reclaiming key markets from it. The open-source community as a whole prepared a major push for mainstream respectability, and began to welcome alliances with major corporations that increasingly feared losing control of their own businesses as Microsoft’s lock-in tactics grew ever bolder.

There was one exception: Richard Stallman and the Free Software Movement. “Open source” was explicitly intended to replace Stallman’s preferred “free software” with a public label that was ideologically neutral, acceptable both to historically opposed groups like the BSD hackers and those who did not wish to take a position in the GPL/anti-GPL debate. Stallman flirted with adopting the term, then rejected it on the grounds that it failed to represent the moral position that was central to his thinking. The Free Software Movement has since insisted on its separateness from “open source”, creating perhaps the most significant political fissure in the hacker culture of 2003.

The other (and more important) intention behind “open source” was to present the hacker community’s methods to the rest of the world (especially the business mainstream) in a more market-friendly, less confrontational way. In this role, fortunately, it proved an unqualified success — and led to a revival of interest in the Unix tradition from which it sprang.

The Lessons of Unix History

The largest-scale pattern in the history of Unix is this: when and where Unix has adhered most closely to open-source practices, it has prospered. Attempts to proprietarize it have invariably resulted in stagnation and decline.

In retrospect, this should probably have become obvious much sooner than it did. We lost ten years after 1984 learning our lesson, and it would probably serve us very ill to ever again forget it.

Being smarter than anyone else about important but narrow issues of software design didn’t prevent us from being almost completely blind about the consequences of interactions between technology and economics that were happening right under our noses. Even the most perceptive and forward-looking thinkers in the Unix community were at best half-sighted. The lesson for the future is that over-committing to any one technology or business model would be a mistake — and maintaining the adaptive flexibility of our software and the design tradition that goes with it is correspondingly imperative.

Another lesson is this: Never bet against the cheap plastic solution. Or, equivalently, the low-end/high-volume hardware technology almost always ends up climbing the power curve and winning. The economist Clayton Christensen calls this *disruptive technology* and showed in *The Innovator’s Dilemma* [Christensen] how this happened with disk drives, steam shovels, and motorcycles. We saw it happen as minicomputers displaced mainframes, workstations and servers replaced minis, and commodity Intel machines replaced workstations and servers. The open-source movement is winning by commoditizing software. To prosper, Unix needs to maintain the knack of co-opting the cheap plastic solution rather than trying to fight it.

Finally, the old-school Unix community failed in its efforts to be “professional” by welcoming in all the command machinery of conventional corporate organization, finance, and marketing. We had to be rescued from our folly by a rebel alliance of obsessive geeks and creative misfits—who then proceeded to show us that professionalism and dedication really meant what we had been doing *before* we succumbed to the mundane persuasions of “sound business practices”.

The application of these lessons with respect to software technologies other than Unix is left as an easy exercise for the reader.

Chapter 3. Contrasts

Comparing the Unix Philosophy with Others

If you have any trouble sounding condescending, find a Unix user to show you how it's done.

--

<author>ScottAdams</author>

Dilbert newsletter 3.0, 1994

The design of operating systems conditions the style of software development under them in many ways both obvious and subtle. Much of this book traces connections between the design of the Unix operating system and the philosophy of program design that has evolved around it. For contrast, it will therefore be instructive to compare the classic Unix way with the styles of design and programming native to other major operating systems.

The Elements of Operating-System Style

Before we can start discussing specific operating systems, we'll need an organizing framework for the ways that operating-system design can affect programming style for good or ill.

Overall, the design and programming styles associated with different operating systems seem to derive from three different sources: (a) the intentions of the operating-system designers, (b) uniformities forced on designs by costs and limitations in the programming environment, and (c) random cultural drift, early practices becoming traditional simply because they were there first.

Even if we take it as given that there is some random cultural drift in every operating-system community, considering the intentions of the designers and the costs and limitations of the results does reveal some interesting patterns that can help us understand the Unix style better by contrast. We can make the patterns explicit by analyzing some of the most important ways that operating systems differ.

What Is the Operating System's Unifying Idea?

Unix has a couple of unifying ideas or metaphors that shape its APIs and the development style that proceeds from them. The most important of these are probably the “everything is a file” model

and the pipe metaphor[20] built on top of it. In general, development style under any given operating system is strongly conditioned by the unifying ideas baked into the system by its designers — they percolate upwards into applications programming from the models provided by system tools and APIs.

Accordingly, the most basic question to ask in contrasting Unix with another operating system is: Does it have unifying ideas that shape its development, and if so how do they differ from Unix's?

To design the perfect anti-Unix, have no unifying idea at all, just an incoherent pile of ad-hoc features.

Multitasking Capability

One of the most basic ways operating systems can differ is in the extent to which they can support multiple concurrent processes. At the lowest end (such as DOS or CP/M) the operating system is basically a sequential program loader with no capacity to multitask at all. Operating systems of this kind are no longer competitive on general-purpose computers.

At the next level up, an operating system may have *cooperative multitasking*. Such systems can support multiple processes, but a process has to voluntarily give up its hold on the processor before the next one can run (thus, simple programming errors can readily freeze the machine). This style of operating system was a transient adaptation to hardware that was powerful enough for concurrency but lacked either a periodic clock interrupt[21] or a memory-management unit or both; it, too, is obsolete and no longer competitive.

Unix has *preemptive multitasking*, in which timeslices are allocated by a scheduler which routinely interrupts or pre-empts the running process in order to hand control to the next one. Almost all modern operating systems support preemption.

Note that "multitasking" is not the same as "multiuser". An operating system can be multitasking but single-user, in which case the facility is used to support a single console and multiple background processes. True multiuser support requires multiple user privilege domains, a feature we'll cover in the discussion of internal boundaries a bit further on.

[20] For readers without Unix experience, a pipe is a way of connecting the output of one program to the input of another. We'll explore the ways this idea can be used to help programs cooperate in Chapter 7.

[21] A periodic clock interrupt from the hardware is useful as a sort of heartbeat for a timesharing system; each time it fires, it tells the system that it may be time to switch to another task, defining the size of the unit timeslice. In 2003 Unixes usually set the heartbeat to either 60 or 100 times a second.

To design the perfect anti-Unix, don't support multitasking at all — or, support multitasking but cripple it by surrounding process management with a lot of restrictions, limitations, and special cases that mean it's quite difficult to get any actual use out of multitasking.

Cooperating Processes

In the Unix experience, inexpensive process-spawning and easy inter-process communication (IPC) makes a whole ecology of small tools, pipes, and filters possible. We'll explore this ecology in Chapter 7; here, we need to point out some consequences of expensive process-spawning and IPC.

> The pipe was technically trivial, but profound in its effect. However, it would not have been trivial without the fundamental unifying notion of the process as an autonomous unit of computation, with process control being programmable. As in Multics, a shell was just another process; process control did not come from God inscribed in JCL.
>
> —
> <author>DougMcIlroy</author>

If an operating system makes spawning new processes expensive and/or process control is difficult and inflexible, you'll usually see all of the following consequences:

- Monster monoliths become a more natural way of programming.
- Lots of policy has to be expressed within those monoliths. This encourages C++ and elaborately layered internal code organization, rather than C and relatively flat internal hierarchies.
- When processes can't avoid a need to communicate, they do so through mechanisms that are either clumsy, inefficient, and insecure (such as temporary files) or by knowing far too much about each others' implementations.
- Multithreading is extensively used for tasks that Unix would handle with multiple communicating lightweight processes.
- Learning and using asynchronous I/O is a must.

These are examples of common stylistic traits (even in applications programming) being driven by a limitation in the OS environment.

A subtle but important property of pipes and the other classic Unix IPC methods is that they require communication between programs to be held down to a level of simplicity that encourages separation of function. Conversely, the result of having no equivalent of the pipe is that programs can only be designed to cooperate by building in full knowledge of each others' internals.

In operating systems without flexible IPC and a strong tradition of using it, programs communicate by sharing elaborate data structures. Because the communication problem has to be solved anew for all programs every time another is added to the set, the complexity of this solution rises as the square of the number of cooperating programs. Worse than that, any change in one of the exposed data structures can induce subtle bugs in an arbitrarily large number of other programs.

> Word and Excel and PowerPoint and other Microsoft programs have intimate — one might say promiscuous — knowledge of each others' internals. In Unix, one tries to design programs to operate not specifically with each other, but with programs as yet unthought of.
>
> —
> <author>DougMcIlroy</author>

We'll return to this theme in Chapter 7.

To design the perfect anti-Unix, make process-spawning very expensive, make process control difficult and inflexible, and leave IPC as an unsupported or half-supported afterthought.

Internal Boundaries

Unix has wired into it an assumption that the programmer knows best. It doesn't stop you or request confirmation when you do dangerous things with your own data, like issuing **rm -rf ***. On the other hand, Unix is rather careful about not letting you step on other people's data. In fact, Unix encourages you to have multiple accounts, each with its own attached and possibly differing privileges, to help you protect yourself from misbehaving programs.[22] System programs often have their own pseudo-user accounts to confer access to special system files without requiring unlimited (or *superuser*) access.

[22]The modern buzzword for this is *role-based security*.

Unix has at least three levels of internal boundaries that guard against malicious users or buggy programs. One is memory management; Unix uses its hardware's memory management unit (MMU) to ensure that separate processes are prevented from intruding on the others' memory-address spaces. A second is the presence of true privilege groups for multiple users — an ordinary (nonroot) user's processes cannot alter or read another user's files without permission. A third is the confinement of security-critical functions to the smallest possible pieces of trusted code. Under Unix, even the shell (the system command interpreter) is not a privileged program.

The strength of an operating system's internal boundaries is not merely an abstract issue of design: It has important practical consequences for the security of the system.

To design the perfect anti-Unix, discard or bypass memory management so that a runaway process can crash, subvert, or corrupt any running program. Have weak or nonexistent privilege groups, so users can readily alter each others' files and the system's critical data (e.g., a macro virus, having seized control of your word processor, can format your hard drive). And trust large volumes of code, like the entire shell and GUI, so that any bug or successful attack on that code becomes a threat to the entire system.

File Attributes and Record Structures

Unix files have neither record structure nor attributes. In some operating systems, files have an associated record structure; the operating system (or its service libraries) knows about files with a fixed record length, or about text line termination and whether CR/LF is to be read as a single logical character.

In other operating systems, files and directories can have name/attribute pairs associated with them — out-of-band data used (for example) to associate a document file with an application that understands it. (The classic Unix way to handle these associations is to have applications recognize 'magic numbers', or other type data within the file itself.)

OS-level record structures are generally an optimization hack, and do little more than complicate APIs and programmers' lives. They encourage the use of opaque record-oriented file formats that generic tools like text editors cannot read properly.

File attributes can be useful, but (as we will see in Chapter 20) can raise some awkward semantic issues in a world of byte-stream-oriented tools and pipes. When file attributes are supported at the operating-system level, they predispose programmers to use opaque formats and lean on the file attributes to tie them to the specific applications that interpret them.

To design the perfect anti-Unix, have a cumbersome set of record structures that make it a hit-or-miss proposition whether any given tool will be able to even read a file as the writer intended it. Add file attributes and have the system depend on them heavily, so that the semantics of a file will not be determinable by looking at the data within it.

Binary File Formats

If your operating system uses binary formats for critical data (such as user-account records) it is likely that no tradition of readable textual formats for applications will develop. We explain in more detail why this is a problem in Chapter 5. For now it's sufficient to note the following consequences:

- Even if a command-line interface, scripting, and pipes are supported, very few filters will evolve.
- Data files will be accessible only through dedicated tools. Developers will think of the tools rather than the data files as central. Thus, different versions of file formats will tend to be incompatible.

To design the perfect anti-Unix, make all file formats binary and opaque, and require heavyweight tools to read and edit them.

Preferred User Interface Style

In Chapter 11 we will develop in some detail the consequences of the differences between *command-line interfaces* (CLIs) and *graphical user interfaces* (GUIs). Which kind an operating system's designers choose as its normal mode of presentation will affect many aspects of the design, from process scheduling and memory management on up to the *application programming interfaces* (APIs) presented for applications to use.

It has been enough years since the first Macintosh that very few people need to be convinced that weak GUI facilities in an operating system are a problem. The Unix lesson is the opposite: that weak CLI facilities are a less obvious but equally severe deficit.

If the CLI facilities of an operating system are weak or nonexistent, you'll also see the following consequences:

- Programs will not be designed to cooperate with each other in unexpected ways — because they *can't* be. Outputs aren't usable as inputs.
- Remote system administration will be sparsely supported, more difficult to use, and more network-intensive.[23]
- Even simple noninteractive programs will incur the overhead of a GUI or elaborate scripting interface.
- Servers, daemons, and background processes will probably be impossible or at least rather difficult, to program in any graceful way.

To design the perfect anti-Unix, have no CLI and no capability to script programs — or, important facilities that the CLI cannot drive.

Intended Audience

The design of operating systems varies in response to the expected audience for the system. Some operating systems are intended for back rooms, some for desktops. Some are designed for technical users, others for end users. Some are intended to work standalone in real-time control applications, others for an environment of timesharing and pervasive networking.

One important distinction is client vs. server. 'Client' translates as: being lightweight, suppporting only a single user, able to run on small machines, designed to be switched on when needed and off when the user is done, lacking pre-emptive multitasking, optimized for low latency, and putting a lot of its resources into fancy user interfaces. 'Server' translates as: being heavyweight, capable of running continuously, optimized for throughput, fully pre-emptively multitasking to handle multiple sessions. In origin all operating systems were server operating systems; the concept of a client operating system only emerged in the late 1970s with inexpensive but underpowered PC hardware. Client operating systems are more focused on a visually attractive user experience than on 24/7 uptime.

All these variables have an effect on development style. One of the most obvious is the level of interface complexity the target audience will tolerate, and how it tends to weight perceived complexity against other variables like cost and capability. Unix is often said to have been written by programmers for programmers — an audience that is notoriously tolerant of interface complexity.

[23] This problem was considered quite serious by Microsoft itself during their rebuild of Hotmail. See [BrooksD].

> This is a consequence rather than a goal. I abhor a system designed for the “user”, if that word is a coded pejorative meaning “stupid and unsophisticated”.
>
> —
>
> <author>KenThompson</author>

To design the perfect anti-Unix, write an operating system that thinks it knows what you’re doing better than you do. And then adds injury to insult by getting it wrong.

Entry Barriers to Development

Another important dimension along which operating systems differ is the height of the barrier that separates mere users from becoming developers. There are two important cost drivers here. One is the monetary cost of development tools, the other is the time cost of gaining proficiency as a developer. Some development cultures evolve social barriers to entry, but these are usually an effect of the underlying technology costs, not a primary cause.

Expensive development tools and complex, opaque APIs produce small and elitist programming cultures. In those cultures, programming projects are large, serious endeavors — they have to be in order to offer a payoff that justifies the cost of both hard and soft (human) capital invested. Large, serious projects tend to produce large, serious programs (and, far too often, large expensive failures).

Inexpensive tools and simple interfaces support casual programming, hobbyist cultures, and exploration. Programming projects can be small (often, formal project structure is plain unnecessary), and failure is not a catastrophe. This changes the style in which people develop code; among other things, they show less tendency to over-commit to failed approaches.

Casual programming tends to produce lots of small programs and a self-reinforcing, expanding community of knowledge. In a world of cheap hardware, the presence or absence of such a community is an increasingly important factor in whether an operating system is long-term viable at all.

Unix pioneered casual programming. One of the things Unix was first at doing was shipping with a compiler and scripting tools as part of the default installation available to all users, supporting a hobbyist software-development culture that spanned multiple installations. Many people who write code under Unix do not think of it as writing code — they think of it as writing scripts to automate common tasks, or as customizing their environment.

To design the perfect anti-Unix, make casual programming impossible.

Operating-System Comparisons

The logic of Unix's design choice stands out more clearly when we contrast it with other operating systems. Here we will attempt only a design overview; for detailed discussion of the technical features of different operating systems.[24]

[24] See the OSData website [http://www.osdata.com/].

Figure 3.1. Schematic history of timesharing.

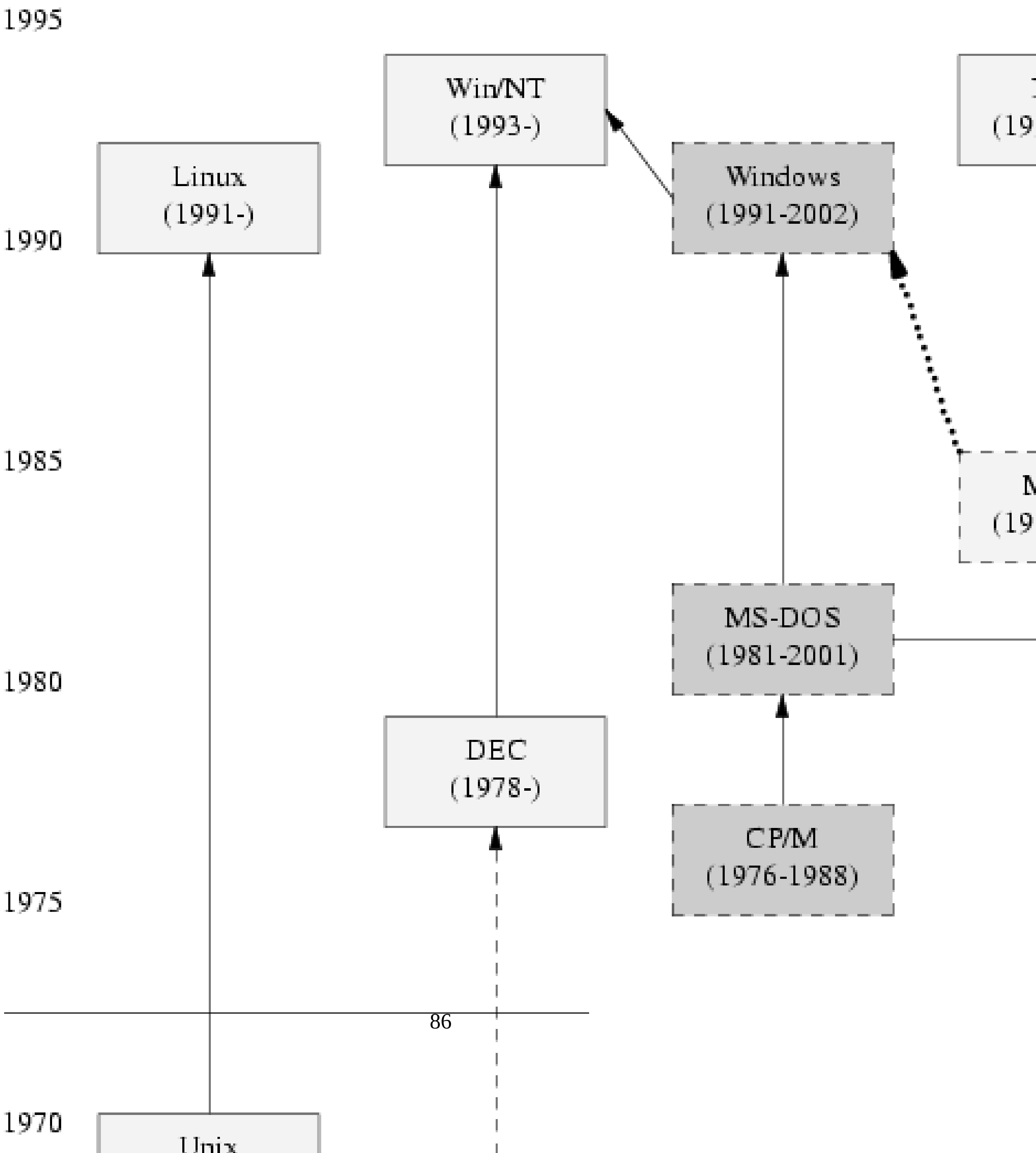


Figure 3.1 indicates the genetic relationships among the timesharing operating systems we'll survey. A few other operating systems (marked in gray, and not necessarily timesharing) are included for context. Sytems in solid boxes are still live. The 'birth' are dates of first shipment;[25] the 'death' dates are generally when the system was end-of-lifed by its vendor.

Solid arrows indicate a genetic relationship or very strong design influence (e.g., a later system with an API deliberately reverse-engineered to match an earlier one). Dashed lines indicate significant design influence. Dotted lines indicate weak design influence. Not all the genetic relationships are acknowledged by the developers; indeed, some have been officially denied for legal or corporate-strategy reasons but are open secrets in the industry.

The 'Unix' box includes all proprietary Unixes, including both AT&T and early Berkeley versions. The 'Linux' box includes the open-source Unixes, all of which launched in 1991. They have genetic inheritance from early Unix through code that was freed from AT&T proprietary control by the settlement of a 1993 lawsuit.[26]

VMS

VMS is the proprietary operating system originally developed for the VAX minicomputer from Digital Equipment Corporation. It was first released in 1978, was an important production operating system in the 1980s and early 1990s, and continued to be maintained when DEC was acquired by Compaq and Compaq was acquired by Hewlett-Packard. It is still sold and supported in mid-2003, though little new development goes on in it today.[27] VMS is surveyed here to show the contrast between Unix and other CLI-oriented operating systems from the minicomputer era.

VMS has full preemptive multitasking, but makes process-spawning very expensive. The VMS file system has an elaborate notion of record types (though not attributes). These traits have all the consequences we outlined earlier on, especially (in VMS's case) the tendency for programs to be huge, clunky monoliths.

VMS features long, readable COBOL-like system commands and command options. It has very comprehensive on-line help (not for APIs, but for the executable programs and command-line syntax). In fact, the VMS CLI and its help system are the organizing metaphor of VMS. Though X

[25] Except for Multics which exerted most of its influence between the time its specifications were published in 1965 and when it actually shipped in 1969.

[26] For details on the lawsuit, see Marshall Kirk McKusick's paper in [OpenSources].

[27] More information is available at the OpenVMS.org site [http://www.openvms.org].

windows has been retrofitted onto the system, the verbose CLI remains the most important stylistic influence on program design. This has the following major implications:

- The frequency with which people use command-line functions — the more voluminously you have to type, the less you want to do it.
- The size of programs — people want to type less, so they want to use fewer programs, and write larger ones with more bundled functions.
- The number and types of options your program accepts — they must conform to the syntactic constraints imposed by the help system.
- The ease of using the help system — it's very complete, but search and discovery tools for it are absent and it has poor indexing. This makes acquiring broad knowledge difficult, encourages specialization, and discourages casual programming.

VMS has a respectable system of internal boundaries. It was designed for true multiuser operation and fully employs the hardware MMU to protect processes from each other. The system command interpreter is privileged, but the encapsulation of critical functions is otherwise reasonably good. Security cracks against VMS have been rare.

VMS tools were initially expensive, and its interfaces are complex. Enormous volumes of VMS programmer documentation are only available in paper form, so looking up anything is a time-consuming, high-overhead operation. This has tended to discourage exploratory programming and learning a large toolkit. Only since being nearly abandoned by its vendor has VMS developed casual programming and a hobbyist culture, and that culture is not particularly strong.

Like Unix, VMS predated the client/server distinction. It was successful in its day as a general-purpose timesharing operating system. The intended audience was primarily technical users and software-intensive businesses, implying a moderate tolerance for complexity.

MacOS

The Macintosh operating system was designed at Apple in the early 1980s, inspired by pioneering work on GUIs done earlier at Xerox's Palo Alto Research Center. It saw its debut with the Macintosh in 1984. MacOS has gone through two significant design transitions since, and is undergoing a third. The first transition was the shift from supporting only a single application at a

time to being able to cooperatively multitask multiple applications (MultiFinder); the second was the shift from 68000 to PowerPC processors, which both preserved backward binary compatibility with 68K applications and brought in an advanced shared library management system for PowerPC applications, replacing the original 68K trap instruction-based code-sharing system. The third was the merger of MacOS design ideas with a Unix-derived infrastructure in MacOS X. Except where specifically noted, the discussion here applies to pre-OS-X versions.

MacOS has a very strong unifying idea that is very different from Unix's: the Mac Interface Guidelines. These specify in great detail what an application GUI should look like and how it should behave. The consistency of the Guidelines influenced the culture of Mac users in significant ways. Not infrequently, simple-minded ports of DOS or Unix programs that did not follow the Guidelines have been summarily rejected by the Mac user base and failed in the marketplace.

One key idea of the Guidelines is that things stay where you put them. Documents, directories, and other objects have persistent locations on the desktop that the system doesn't mess with, and the desktop context persists through reboots.

The Macintosh's unifying idea is so strong that most of the other design choices we discussed above are either forced by it or invisible. All programs have GUIs. There is no CLI at all. Scripting facilities are present but much less commonly used than under Unix; many Mac programmers never learn them. MacOS's captive-interface GUI metaphor (organized around a single main event loop) leads to a weak scheduler without preemption. The weak scheduler, and the fact that all MultiFinder applications run in a single large address space, implies that it is not practical to use separated processes or even threads rather than polling.

MacOS applications are not, however, invariably monster monoliths. The system's GUI support code, which is partly implemented in a ROM shipped with the hardware and partly implemented in shared libraries, communicates with MacOS programs through an event interface that has been quite stable since its beginnings. Thus, the design of the operating system encourages a relatively clean separation between application engine and GUI interface.

MacOS also has strong support for isolating application metadata like menu structures from the engine code. MacOS files have both a 'data fork' (a Unix-style bag of bytes that contains a document or program code) and a 'resource fork' (a set of user-definable file attributes). Mac applications tend to be designed so that (for example) the images and sound used in them are stored in the resource fork and can be modified separately from the application code.

The MacOS system of internal boundaries is very weak. There is a wired-in assumption that there is but a single user, so there are no per-user privilege groups. Multitasking is cooperative, not pre-emptive. All MultiFinder applications run in the same address space, so bad code in any application can corrupt anything outside the operating system's low-level kernel. Security cracks against MacOS machines are very easy to write; the OS has been spared an epidemic mainly because very few people are motivated to crack it.

Mac programmers tend to design in the opposite direction from Unix programmers; that is, they work from the interface inward, rather than from the engine outward (we'll discuss some of the implications of this choice in Chapter 20). Everything in the design of the MacOS conspires to encourage this.

The intended role for the Macintosh was as a client operating system for nontechnical end users, implying a very low tolerance for interface complexity. Developers in the Macintosh culture became very, very good at designing simple interfaces.

The incremental cost of becoming a developer, assuming you have a Macintosh already, has never been high. Thus, despite rather complex interfaces, the Mac developed a strong hobbyist culture early on. There is a vigorous tradition of small tools, shareware, and user-supported software.

Classic MacOS has been end-of-lifed. Most of its facilities have been imported into MacOS X, which mates them to a Unix infrastructure derived from the Berkeley tradition.[28] At the same time, leading-edge Unixes such as Linux are beginning to borrow ideas like file attributes (a generalization of the resource fork) from MacOS.

OS/2

OS/2 began life as an IBM development project called ADOS ('Advanced DOS'), one of three competitors to become DOS 4. At that time, IBM and Microsoft were formally collaborating to develop a next-generation operating system for the PC. OS/2 1.0 was first released in 1987 for the 286, but was unsuccessful. The 2.0 version for the 386 came out in 1992, but by that time the IBM/Microsoft alliance had already fractured. Microsoft went in a different (and more lucrative) direction with Windows 3.0. OS/2 attracted a loyal minority following, but never attracted a critical mass of developers and users. It remained third in the desktop market, behind the Macintosh, until being subsumed into IBM's Java initiative after 1996. The last released version was 4.0 in 1996.

[28] MacOS X actually consists of two proprietary layers (ports of the OpenStep and Classic Mac GUIs) layered over an open-source Unix core (Darwin).

Early versions found their way into embedded systems and still, as of mid-2003, run inside many of the world's automated teller machines.

Like Unix, OS/2 was built to be preemptively multitasking and would not run on a machine without an MMU (early versions simulated an MMU using the 286's memory segmentation). Unlike Unix, OS/2 was never built to be a multiuser system. Process-spawning was relatively cheap, but IPC was difficult and brittle. Networking was initially focused on LAN protocols, but a TCP/IP stack was added in later versions. There were no programs analogous to Unix service daemons, so OS/2 never handled multi-function networking very well.

OS/2 had both a CLI and GUI. Most of the positive legendry around OS/2 was about the Workplace Shell (WPS), the OS/2 desktop. Some of this technology was licensed from the developers of the AmigaOS Workbench,[29] a pioneering GUI desktop that still as of 2003 has a loyal fan base in Europe. This is the one area of the design in which OS/2 achieved a level of capability which Unix arguably has not yet matched. The WPS was a clean, powerful, object-oriented design with understandable behavior and good extensibility. Years later it would become a model for Linux's GNOME project.

The class-hierarchy design of WPS was one of OS/2's unifying ideas. The other was multithreading. OS/2 programmers used threading heavily as a partial substitute for IPC between peer processes. No tradition of cooperating program toolkits developed.

OS/2 had the internal boundaries one would expect in a single-user OS. Running processes were protected from each other, and kernel space was protected from user space, but there were no per-user privilege groups. This meant the file system had no protection against malicious code. Another consequence was that there was no analog of a home directory; application data tended to be scattered all over the system.

A further consequence of the lack of multiuser capability was that there could be no privilege distinctions in userspace. Thus, developers tended to trust only kernel code. Many system tasks that in Unix would be handled by user-space daemons were jammed into the kernel or the WPS. Both bloated as a result.

OS/2 had a text vs. binary mode (that is, a mode in which CR/LF was read as a single end-of-line, versus one in which no such interpretation was performed), but no other file record structure. It supported file attributes, which were used for desktop persistence after the manner of the Macintosh. System databases were mostly in binary formats.

[29] In return for some Amiga technology, IBM gave Commodore a license for its REXX scripting language. The deal is described at http://www.os2bbs.com/os2news/OS2Warp.html.

The preferred UI style was through the WPS. User interfaces tended to be ergonomically better than Windows, though not up to Macintosh standards (OS/2's most active period was relatively early in the history of MacOS Classic). Like Unix and Windows, OS/2's user interface was themed around multiple, independent per-task groups of windows, rather than capturing the desktop for the running application.

The intended audience for OS/2 was business and nontechnical end users, implying a low tolerance for interface complexity. It was used both as a client operating system and as a file and print server.

In the early 1990s, developers in the OS/2 community began to migrate to a Unix-inspired environment called EMX that emulated POSIX interfaces. Ports of Unix software started routinely showing up under OS/2 in the latter half of the 1990s.

Anyone could download EMX, which included the GNU Compiler Collection and other open-source development tools. IBM intermittently gave away copies of the system documentation in the OS/2 developer's toolkit, which was posted on many BBSs and FTP sites. Because of this, the "Hobbes" FTP archive of user-developed OS/2 software had already grown to over a gigabyte in size by 1995. A very vigorous tradition of small tools, exploratory programming, and shareware developed and retained a loyal following for some years after OS/2 itself was clearly headed for the dustbin of history.

After the release of Windows 95 the OS/2 community, feeling beleaguered by Microsoft and encouraged by IBM, became increasingly interested in Java. After the Netscape source code release in early 1998, the direction of migration changed (rather suddenly), toward Linux.

OS/2 is interesting as a case study in how far a multitasking but single-user operating-system design can be pushed. Most of the observations in this case study would apply well to other operating systems of the same general type, notably AmigaOS[30] and GEM.[31] A wealth of OS/2 material is still available on the Web in 2003, including some good histories.[32]

Windows NT

Windows NT (New Technology) is Microsoft's operating system for high-end personal and server use; it is shipped in several variants that can all be considered the same for our purposes. All of Microsoft's operating systems since the demise of Windows ME in 2000 have been NT-based;

[30] AmigaOS Portal [http://os.amiga.com/].
[31] The GEM Operating System [http://www.geocities.com/SiliconValley/Vista/6148/gem.html].
[32] See, for example, the OS Voice [http://www.os2voice.org/] and OS/2 BBS.COM [http://www.os2bbs.com/] sites.

Windows 2000 was NT 5, and Windows XP (current in 2003) is NT 5.1. NT is genetically descended from VMS, with which it shares some important characteristics.

NT has grown by accretion, and lacks a unifying metaphor corresponding to Unix's "everything is a file" or the MacOS desktop.[33] Because core technologies are not anchored in a small set of persistent central metaphors, they become obsolete every few years. Each of the technology generations — DOS (1981), Windows 3.1 (1992), Windows 95 (1995), Windows NT 4 (1996), Windows 2000 (2000), Windows XP (2002), and Windows Server 2003 (2003) — has required that developers relearn fundamental things in a different way, with the old way declared obsolete and no longer well supported.

There are other consequences as well:

- The GUI facilities coexist uneasily with the weak, remnant command-line interface inherited from DOS and VMS.
- Socket programming has no unifying data object analogous to the Unix everything-is-a-file-handle, so multiprogramming and network applications that are simple in Unix require several more fundamental concepts in NT.

NT has file attributes in some of its file system types. They are used in a restricted way, to implement access-control lists on some file systems, and don't affect development style very much. It also has a record-type distinction, between text and binary files, that produces occasional annoyances (both NT and OS/2 inherited this misfeature from DOS).

Though pre-emptive multitasking is supported, process-spawning is expensive — not as expensive as in VMS, but (at about 0.1 seconds per spawn) up to an order of magnitude more so than on a modern Unix. Scripting facilities are weak, and the OS makes extensive use of binary file formats. In addition to the expected consequences we outlined earlier are these:

- Most programs cannot be scripted at all. Programs rely on complex, fragile *remote procedure call* (RPC) methods to communicate with each other, a rich source of bugs.

[33] Perhaps. It has been argued that the unifying metaphor of all Microsoft operating systems is "the customer must be locked in".

- There are no generic tools at all. Documents and databases can't be read or edited without special-purpose programs.
- Over time, the CLI has become more and more neglected because the environment there is so sparse. The problems associated with a weak CLI have gotten progressively worse rather than better. (Windows Server 2003 attempts to reverse this trend somewhat.)

System and user configuration data are centralized in a central properties registry rather than being scattered through numerous dotfiles and system data files as in Unix. This also has consequences throughout the design:

- The registry makes the system completely non-orthogonal. Single-point failures in applications can corrupt the registry, frequently making the entire operating system unusable and requiring a reinstall.
- The *registry creep* phenomenon: as the registry grows, rising access costs slow down all programs.

NT systems on the Internet are notoriously vulnerable to worms, viruses, defacements, and cracks of all kinds. There are many reasons for this, some more fundamental than others. The most fundamental is that NT's internal boundaries are extremely porous.

NT has access control lists that can be used to implement per-user privilege groups, but a great deal of legacy code ignores them, and the operating system permits this in order not to break backward compatibility. There are no security controls on message traffic between GUI clients, either,[34] and adding them would also break backward compatibility.

While NT will use an MMU, NT versions after 3.5 have the system GUI wired into the same address space as the privileged kernel for performance reasons. Recent versions even wire the webserver into kernel space in an attempt to match the speed of Unix-based webservers.

These holes in the boundaries have the synergistic effect of making actual security on NT systems effectively impossible.[35] If an intruder can get code run as any user at all (e.g., through the Outlook email-macro feature), that code can forge messages through the window system to any other running

[34] http://security.tombom.co.uk/shatter.html

[35] Microsoft actually admitted publicly that NT security is impossible in March 2003. See http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS03-010.asp.

application. And any buffer overrun or crack in the GUI or webserver can be exploited to take control of the entire system.

Because Windows does not handle library versioning properly, it suffers from a chronic configuration problem called “DLL hell”, in which installing new programs can randomly upgrade (or even downgrade!) the libraries on which existing programs depend. This applies to the vendor-supplied system libraries as well as to application-specific ones: it is not uncommon for an application to ship with specific versions of system libraries, and break silently when it does not have them.[36]

On the bright side, NT provides sufficient facilities to host Cygwin, which is a compatibility layer implementing Unix at both the utilities and the API level, with remarkably few compromises.[37] Cygwin permits C programs to make use of both the Unix and the native APIs, and is the first thing many Unix hackers install on such Windows systems as they are compelled by circumstances to make use of.

The intended audience for the NT operating systems is primarily nontechnical end users, implying a very low tolerance for interface complexity. It is used in both client and server roles.

Early in its history Microsoft relied on third-party development to supply applications. They originally published full documentation for the Windows APIs, and kept the price of development tools low. But over time, and as competitors collapsed, Microsoft’s strategy shifted to favor in-house development, they began hiding APIs from the outside world, and development tools grew more expensive. As early as Windows 95, Microsoft was requiring nondisclosure agreements as a condition for purchasing professional-quality development tools.

The hobbyist and casual-developer culture that had grown up around DOS and earlier Windows versions was large enough to be self-sustaining even in the face of increasing efforts by Microsoft to lock them out (including such measures as certification programs designed to delegitimize amateurs). Shareware never went away, and Microsoft’s policy began to reverse somewhat after 2000 under market pressure from open-source operating systems and Java. However, Windows interfaces for ‘professional’ programming continued to grow more complex over time, presenting an increasing barrier to casual (or serious!) coding.

[36]The DLL hell problem is somewhat mitigated by the .NET development framework, which handles library versioning — but as of 2003 .NET only ships on the highest-end server versions of NT.

[37]Cygwin is largely compliant with the Single Unix Specification, but programs requiring direct hardware access run into limitations in the Windows kernel that hosts it. Ethernet cards are notoriously problematic.

The result of this history is a sharp dichotomy between the design styles practiced by amateur and professional NT developers — the two groups barely communicate. While the hobbyist culture of small tools and shareware is very much alive, professional NT projects tend to produce monster monoliths even bulkier than those characteristic of 'elitist' operating systems like VMS.

Unix-like shell facilities, command sets, and library APIs are available under Windows through third-party libraries including UWIN, Interix, and the open-source Cygwin.

BeOS

Be, Inc. was founded in 1989 as a hardware vendor, building pioneering multiprocessing machines around the PowerPC chip. BeOS was Be's attempt to add value to the hardware by inventing a new, network-ready operating system model incorporating the lessons of both Unix and the MacOS family, without being either. The result was a tasteful, clean, and exciting design with excellent performance in its chosen role as a multimedia platform.

BeOS's unifying ideas were 'pervasive threading', multimedia flows, and the file system as database. BeOS was designed to minimize latency in the kernel, making it well-suited for processing large volumes of data such as audio and video streams in real time. BeOS 'threads' were actually lightweight processes in Unix terminology, since they supported thread-local storage and therefore did not necessarily share all address spaces. IPC via shared memory was fast and efficient.

BeOS followed the Unix model in having no file structure above the byte level. Like the MacOS, it supported and used file attributes. In fact, the BeOS file system was actually a database that could be indexed by any attribute.

One of the things BeOS took from Unix was intelligent design of internal boundaries. It made full use of an MMU, and sealed running processes off from each other effectively. While it presented as a single-user operating system (no login), it supported Unix-like privilege groups in the file system and elsewhere in the OS internals. These were used to protect system-critical files from being touched by untrusted code; in Unix terms, the user was logged in as an anonymous guest at boot time, with the only other 'user' being root. Full multiuser operation would have been a small change to the upper levels of the system, and there was in fact a BeLogin utility.

BeOS tended to use binary file formats and the native database built into the file system, rather than Unix-like textual formats.

The preferred UI style of BeOS was GUI, and it leaned heavily on MacOS experience in interface design. CLI and scripting were, however, also fully supported. The command-line shell of BeOS was a port of bash(1), the dominant open-source Unix shell, running through a POSIX compatibility library. Porting of Unix CLI software was, by design, trivially easy. Infrastructure to support the full panoply of scripting, filters, and service daemons that goes with the Unix model was in place.

BeOS's intended role was as a client operating system specialized for near-real-time multimedia processing (especially sound and video manipulation). Its intended audience included technical and business end users, implying a moderate tolerance for interface complexity.

Entry barriers to BeOS development were low; though the operating system was proprietary, development tools were inexpensive and full documentation was readily available. The BeOS effort began as part of one of the efforts to unseat Intel's hardware with RISC technology, and was continued as a software-only effort after the Internet explosion. Its strategists were paying attention during Linux's formative period in the early 1990s, and were fully aware of the value of a large casual-developer base. In fact they succeeded in attracting an intensely loyal following; as of 2003 no fewer than five separate projects are attempting to resurrect BeOS in open source.

Unfortunately, the business strategy surrounding BeOS was not as astute as the technical design. The BeOS software was originally bundled with dedicated hardware, and marketed with only vague hints about intended applications. Later (1998) BeOS was ported to generic PCs and more closely focused on multimedia applications, but never attracted a critical mass of applications or users. BeOS finally succumbed in 2001 to a combination of anticompetitive maneuvering by Microsoft (lawsuit in progress as of 2003) and competition from variants of Linux that had been adapted for multimedia handling.

MVS

MVS (Multiple Virtual Storage) is IBM's flagship operating system for its mainframe computers. Its roots stretch back to OS/360, which began life in the mid-1960s as the operating system IBM wanted its customers to use on the then-new System/360 computer systems. Descendants of this code remain at the heart of today's IBM mainframe operating systems. Though the code has been almost entirely rewritten, the basic design is largely untouched; backward compatibility has been religiously maintained, to the point that applications written for OS/360 run unmodified on the MVS of 64-bit z/Series mainframe computers three architectural generations later.

Of all the operating systems surveyed here, MVS is the only one that could be considered older than Unix (the ambiguity stems from the degree to which it has evolved over time). It is also the least

influenced by Unix concepts and technology, and represents the strongest design contrast with Unix. The unifying idea of MVS is that all work is batch; the system is designed to make the most efficient possible use of the machine for batch processing of huge amounts of data, with minimal concessions to interaction with human users.

Native MVS terminals (the 3270 series) operate only in block mode. The user is presented with a screen that he fills in, modifying local storage in the terminal. No interrupt is presented to the mainframe until the user presses the send key. Character-level interaction, in the manner of Unix's raw mode, is impossible.

TSO, the closest equivalent to the Unix interactive environment, is limited in native capabilities. Each TSO user is represented to the rest of the system as a simulated batch job. The facility is expensive — so much so that its use is typically limited to programmers and support staff. Ordinary users who need to merely run applications from a terminal almost never use TSO. Instead, they work through transaction monitors, a kind of multiuser application server that does cooperative multitasking and supports asynchronous I/O. In effect, each kind of transaction monitor is a specialized timesharing plugin (almost, but not entirely unlike a webserver running CGI).

Another consequence of the batch-oriented architecture is that process spawning is a slow operation. The I/O system deliberately trades high setup cost (and associated latency) for better throughput. These choices are a good match for batch operation, but deadly to interactive response. A predictable result is that TSO users nowadays spend almost all their time inside a dialog-driven interactive environment, ISPF. It is rare for a programmer to do anything inside native TSO except start up an instance of ISPF. This does away with process-spawn overhead, at the cost of introducing a very large program that does everything but start the machine room coffeepot.

MVS uses the machine MMU; processes have separate address spaces. Interprocess communication is supported only through shared memory. There are facilities for threading (which MVS calls "subtasking"), but they are lightly used, mainly because the facility is only easily accessible from programs written in assembler. Instead, the typical batch application is a short series of heavyweight program invocations glued together by JCL (Job Control Language) which provides scripting, though in a notoriously difficult and inflexible way. Programs in a job communicate through temporary files; filters and the like are nearly impossible to do in a usable manner.

Every file has a record format, sometimes implied (inline input files in JCL are implied to have an 80-byte fixed-length record format inherited from punched cards, for example), but more often explicitly specified. Many system configuration files are in text format, but application files are

usually in binary formats specific to the application. Some general tools for examining files have evolved out of sheer necessity, but it is still not an easy problem to solve.

File system security was an afterthought in the original design. However, when security was found to be necessary, IBM added it in an inspired fashion: They defined a generic security API, then made all file access requests pass by that interface before being processed. As a result, there are at least three competing security packages with differing design philosophies — and all of them are quite good, with no known cracks against them between 1980 and mid-2003. This variety allows an installation to select the package that best suits local security policy.

Networking facilities are another afterthought. There is no concept of one interface for both network connections and local files; their programming interfaces are separate and quite different. This did allow TCP/IP to supplant IBM's native SNA (Systems Network Architecture) as the network protocol of choice fairly seamlessly. It is still common in 2003 to see both in use at a given installation, but SNA is dying out.

Casual programming for MVS is almost nonexistent except within the community of large enterprises that run MVS. This is not due so much to the cost of the tools themselves as it is to the cost of the environment — when one must spend several million dollars on the computer system, a few hundred dollars a month for a compiler is almost incidental. Within that community, however, there is a thriving culture of freely available software, mainly programming and system-administration tools. The first computer user's group, SHARE, was founded in 1955 by IBM users, and is still going strong today.

Considering the vast architectural differences, it is a remarkable fact that MVS was the first non-System-V operating system to meet the Single Unix Specification (there is less to this than meets the eye, however, as ports of Unix software from elsewhere have a strong tendency to founder on ASCII-vs.-EBCDIC character-set issues). It's possible to start a Unix shell from TSO; Unix file systems are specially formatted MVS data sets. The MVS Unix character set is a special EBCDIC codepage with newline and linefeed swapped (so that what appears as linefeed to Unix appears like newline to MVS), but the system calls are real system calls implemented in the MVS kernel.

As the cost of the environment drops into the hobbyist range, there is a small but growing group of users of the last public-domain version of MVS (3.8, dating from 1979). This system, as well as development tools and the emulator to run them, are all available for the cost of a CD.[38]

[38] http://www.cbttape.org/cdrom.htm

The intended role of MVS has always been in the back office. Like VMS and Unix itself, MVS predates the server/client distinction. Interface complexity for back-office users is not only tolerated but expected, in the name of making the computer spend fewer expensive resources on interfaces and more on the work it's there to get done.

VM/CMS

VM/CMS is IBM's *other* mainframe operating system. Historically speaking, it is Unix's uncle: the common ancestor is the CTSS system, developed at MIT around 1963 and running on the IBM 7094 mainframe. The group that developed CTSS then went on to write Multics, the immediate ancestor of Unix. IBM established a group in Cambridge to write a timesharing system for the IBM 360/40, a modified 360 with (for the first time on an IBM system) a paging MMU.[39] The MIT and IBM programmers continued to interact for many years thereafter, and the new system got a user interface that was very CTSS-like, complete with a shell named EXEC and a large supply of utilities analogous to those used on Multics and later on Unix.

In another sense, VM/CMS and Unix are funhouse mirror images of one another. The unifying idea of the system, provided by the VM component, is virtual machines, each of which looks exactly like the underlying physical machine. They are preemptively multitasked, and run either the single-user operating system CMS or a complete multitasking operating system (typically MVS, Linux, or another instance of VM itself). Virtual machines correspond to Unix processes, daemons, and emulators, and communication between them is accomplished by connecting the virtual card punch of one machine to the virtual card reader of another. In addition, a layered tools environment called CMS Pipelines is provided within CMS, directly modeled on Unix's pipes but architecturally extended to support multiple inputs and outputs.

When communication between them has not been explicitly set up, virtual machines are completely isolated from each other. The operating system has the same high reliability, scalability, and security as MVS, and has far greater flexibility and is much easier to use. In addition, the kernel-like portions of CMS do not need to be trusted by the VM component, which is maintained completely separately.

Although CMS is record-oriented, the records are essentially equivalent to the lines that Unix textual tools use. Databases are much better integrated into CMS Pipelines than is typically the case on Unix, where most databases are quite separate from the operating system. In recent years, CMS has been augmented to fully support the Single Unix Specification.

[39] The development machine and initial target was a 40 with customized microcode, but it proved insufficiently powerful; production deployment was on the 360/67.

The UI style of CMS is interactive and conversational, very unlike MVS but like VMS and Unix. A full-screen editor called XEDIT is heavily used.

VM/CMS predates the client/server distinction, and is nowadays used almost entirely as a server operating system with emulated IBM terminals. Before Windows came to dominate the desktop so completely, VM/CMS provided word-processing services and email both internally to IBM and between mainframe customer sites — indeed, many VM systems were installed exclusively to run those applications because of VM's ready scalability to tens of thousands of users.

A scripting language called Rexx supports programming in a style not unlike shell, awk, Perl or Python. Consequently, casual programming (especially by system administrators) is very important on VM/CMS. Free cycles permitting, admins often prefer to run production MVS in a virtual machine rather than directly on the bare iron, so that CMS is also available and its flexibility can be taken advantage of. (There are CMS tools that permit access to MVS file systems.)

There are even striking parallels between the history of VM/CMS within IBM and Unix within Digital Equipment Corporation (which made the hardware that Unix first ran on). It took IBM years to understand the strategic importance of its unofficial timesharing system, and during that time a community of VM/CMS programmers arose that was closely analogous in behavior to the early Unix community. They shared ideas, shared discoveries about the system, and above all shared source code for utilities. No matter how often IBM tried to declare VM/CMS dead, the community — which included IBM's own MVS system developers! — insisted on keeping it alive. VM/CMS even went through the same cycle of de facto open source to closed source back to open source, though not as thoroughly as Unix did.

What VM/CMS lacks, however, is any real analog to C. Both VM and CMS were written in assembler and have remained so implemented. The nearest equivalent to C was various cut-down versions of PL/I that IBM used for systems programming, but did not share with its customers. Therefore, the operating system remains trapped on its original architectural line, though it has grown and expanded as the 360 architecture became the 370 series, the XA series, and finally the current z/Series.

Since the year 2000, IBM has been promoting VM/CMS on mainframes to an unprecedented degree — as ways to host thousands of virtual Linux machines at once.

Linux

Linux, originated by Linus Torvalds in 1991, leads the pack of new-school open-source Unixes that have emerged since 1990 (also including FreeBSD, NetBSD, OpenBSD, and Darwin), and is representative of the design direction being taken by the group as a whole. The trends in it can be taken as typical for this entire group.

Linux does not include any code from the original Unix source tree, but it was designed from Unix standards to behave like a Unix. In the rest of this book, we emphasize the continuity between Unix and Linux. That continuity is extremely strong, both in terms of technology and key developers — but here we emphasize some directions Linux is taking that mark a departure from 'classical' Unix tradition.

Many developers and activists in the Linux community have ambitions to win a substantial share of end-user desktops. This makes Linux's intended audience quite a bit broader than was ever the case for the old-school Unixes, which have primarily aimed at the server and technical-workstation markets. This has implications for the way Linux hackers design software.

The most obvious change is a shift in preferred interface styles. Unix was originally designed for use on teletypes and slow printing terminals. Through much of its lifetime it was strongly associated with character-cell video-display terminals lacking either graphics or color capabilities. Most Unix programmers stayed firmly wedded to the command line long after large end-user applications had migrated to X-based GUIs, and the design of both Unix operating systems and their applications have continued to reflect this fact.

Linux users and developers, on the other hand, have been adapting themselves to address the nontechnical user's fear of CLIs. They have moved to building GUIs and GUI tools much more intensively than was the case in old-school Unix, or even in contemporary proprietary Unixes. To a lesser but significant extent, this is true of the other open-source Unixes as well.

The desire to reach end users has also made Linux developers much more concerned with smoothness of installation and software distribution issues than is typically the case under proprietary Unix systems. One consequence is that Linux features binary-package systems far more sophisticated than any analogs in proprietary Unixes, with interfaces designed (as of 2003, with only mixed success) to be palatable to nontechnical end users.

The Linux community wants, more than the old-school Unixes ever did, to turn their software into a sort of universal pipefitting for connecting together other environments. Thus, Linux features support for reading and (often) writing the file system formats and networking methods native to other operating systems. It also supports multiple-booting with them on the same hardware, and

simulating them in software inside Linux itself. The long-term goal is subsumption; Linux emulates so it can absorb.[40]

The goal of subsuming the competition, combined with the drive to reach the end-user, has motivated Linux developers to adopt design ideas from non-Unix operating systems to a degree that makes traditional Unixes look rather insular. Linux applications using Windows .INI format files for configuration is a minor example we'll cover in Chapter 10; Linux 2.5's incorporation of extended file attributes, which among other things can be used to emulate the semantics of the Macintosh resource fork, is a recent major one at time of writing.

> But the day Linux gives the Mac diagnostic that you can't open a file because you don't have the application is the day Linux becomes non-Unix.
>
> —
>
> <author>DougMcIlroy</author>

The remaining proprietary Unixes (such as Solaris, HP-UX, AIX, etc.) are designed to be big products for big IT budgets. Their economic niche encourages designs optimized for maximum power on high-end, leading-edge hardware. Because Linux has part of its roots among PC hobbyists, it emphasizes doing more with less. Where proprietary Unixes are tuned for multiprocessor and server-cluster operation at the expense of performance on low-end hardware, core Linux developers have explicitly chosen not to accept more complexity and overhead on low-end machines for marginal performance gains on high-end hardware.

Indeed, a substantial fraction of the Linux user community is understood to be wringing usefulness out of hardware as technically obsolete today as Ken Thompson's PDP-7 was in 1969. As a consequence, Linux applications are under pressure to stay lean and mean that their counterparts under proprietary Unix do not experience.

These trends have implications for the future of Unix as a whole, a topic we'll return to in Chapter 20.

What Goes Around, Comes Around

We attempted to select for comparison timesharing systems that either are now or have in the past been competitive with Unix. The field of plausible candidates is not wide. Most (Multics, ITS, DTSS, TOPS-10, TOPS-20, MTS, GCOS, MPE and perhaps a dozen others) are so long dead that

[40] The results of Linux's emulate-and-subsume strategy differ noticeably from the embrace-and-extend practiced by some of its competitors. For starters, Linux does not break compatibility with what it is emulating in order to lock customers into the "extended" version.

they are fading from the collective memory of the computing field. Of those we surveyed, VMS and OS/2 are moribund, and MacOS has been subsumed by a Unix derivative. MVS and VM/CMS were limited to a single proprietary mainframe line. Only Microsoft Windows remains as a viable competitor independent of the Unix tradition.

We pointed out Unix's strengths in Chapter 1, and they are certainly part of the explanation. But it's actually more instructive to look at the obverse of that answer and ask which weaknesses in Unix's competitors did them in.

The most obvious shared problem is nonportability. Most of Unix's pre-1980 competitors were tied to a single hardware platform, and died with that platform. One reason VMS survived long enough to merit inclusion here as a case study is that it was successfully ported from its original VAX hardware to the Alpha processor (and in 2003 is being ported from Alpha to Itanium). MacOS successfully made the jump from the Motorola 68000 to PowerPC chips in the late 1980s. Microsoft Windows escaped this problem by being in the right place when commoditization flattened the market for general-purpose computers into a PC monoculture.

From 1980 on, another particular weakness continually reemerges as a theme in different systems that Unix either steamrollered or outlasted: an inability to support networking gracefully.

In a world of pervasive networking, even an operating system designed for single-user use needs multiuser capability (multiple privilege groups) — because without that, any network transaction that can trick a user into running malicious code will subvert the entire system (Windows macro viruses are only the tip of this iceberg). Without strong multitasking, the ability of an operating system to handle network traffic and run user programs at the same time will be impaired. The operating system also needs efficient IPC so that its network programs can communicate with each other and with the user's foreground applications.

Windows gets away with having severe deficiencies in these areas only by virtue of having developed a monopoly position before networking became really important, and by having a user population that has been conditioned to accept a shocking frequency of crashes and security breaches as normal. This is not a stable situation, and it is one that partisans of Linux have successfully (in 2003) exploited to make major inroads in the server-operating-system market.

Around 1980, during the early heyday of personal computers, operating-system designers dismissed Unix and traditional timesharing as heavyweight, cumbersome, and inappropriate for the brave new world of single-user personal machines — despite the fact that GUI interfaces tended to demand the reinvention of multitasking to cope with threads of execution bound to different windows and

widgets. The trend toward client operating systems was so intense that server operating systems were at times dismissed as steam-powered relics of a bygone age.

But as the designers of BeOS noticed, the requirements of pervasive networking cannot be met without implementing something very close to general-purpose timesharing. Single-user client operating systems cannot thrive in an Internetted world.

This problem drove the reconvergence of client and server operating systems. The first, pre-Internet attempts at peer-to-peer networking over LANs, in the late 1980s, began to expose the inadequacy of the client-OS design model. Data on a network has to have rendezvous points in order to be shared; thus, we can't do without servers. At the same time, experience with the Macintosh and Windows client operating systems raised the bar on the minimum quality of user experience customers would tolerate.

With non-Unix models for timesharing effectively dead by 1990, there were not many possible responses client operating-system designers could mount to this challenge. They could co-opt Unix (as MacOS X has done), re-invent roughly equivalent features a patch at a time (as Windows has done), or attempt to reinvent the entire world (as BeOS tried and failed to do). But meanwhile, open-source Unixes were growing client-like capabilities to use GUIs and run on inexpensive personal machines.

These pressures turned out, however, not to be as symmetrically balanced as the above description might imply. Retrofitting server-operating-system features like multiple privilege classes and full multitasking onto a client operating system is very difficult, quite likely to break compatibility with older versions of the client, and generally produces a fragile and unsatisfactory result rife with stability and security problems. Retrofitting a GUI onto a server operating system, on the other hand, raises problems that can largely be finessed by a combination of cleverness and throwing ever-more-inexpensive hardware resources at the problem. As with buildings, it's easier to repair superstructure on top of a solid foundation than it is to replace the foundations without trashing the superstructure.

Besides having the native architectural strengths of a server operating system, Unix was always agnostic about its intended audience. Its designers and implementers never assumed they knew all potential uses the system would be put to.

Thus, the Unix design proved more capable of reinventing itself as a client than any of its client-operating-system competitors were of reinventing themselves as servers. While many other factors of technology and economics contributed to the Unix resurgence of the 1990s, this is one that really foregrounds itself in any discussion of operating-system design style.

Part II. Design

Chapter 4. Modularity

Keeping It Clean, Keeping It Simple

There are two ways of constructing a software design. One is to make it so simple that there are obviously no deficiencies; the other is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

--

<author>C.A. R.Hoare</author>

The Emperor's Old Clothes, CACM February 1981

There is a natural hierarchy of code-partitioning methods that has evolved as programmers have had to manage ever-increasing levels of complexity. In the beginning, everything was one big lump of machine code. The earliest procedural languages brought in the notion of partition by subroutine. Then we invented service libraries to share common utility functions among multiple programs. Next, we invented separated address spaces and communicating processes. Today we routinely distribute program systems across multiple hosts separated by thousands of miles of network cable.

The early developers of Unix were among the pioneers in software modularity. Before them, the Rule of Modularity was computer-science theory but not engineering practice. In *Design Rules* [Baldwin-Clark], a path-breaking study of the economics of modularity in engineering design, the authors use the development of the computer industry as a case study and argue that the Unix community was in fact the first to systematically apply modular decomposition to production software, as opposed to hardware. Modularity of hardware has of course been one of the foundations of engineering since the adoption of standard screw threads in the late 1800s.

The Rule of Modularity bears amplification here: The only way to write complex software that won't fall on its face is to build it out of simple modules connected by well-defined interfaces, so that most problems are local and you can have some hope of fixing or optimizing a part without breaking the whole.

The tradition of being careful about modularity and of paying close attention to issues like orthogonality and compactness are still much deeper in the bone among Unix programmers than elsewhere.

> Early Unix programmers became good at modularity because they had to be. An OS is one of the most complicated pieces of code around. If it is not well structured, it will fall apart. There were a couple of early failures at building Unix that were scrapped. One can blame the early (structureless) C for this, but

> basically it was because the OS was too complicated to write. We needed both refinements in tools (like C structures) and good practice in using them (like Rob Pike's rules for programming) before we could tame that complexity.
>
> —
>
> <author>KenThompson</author>

Early Unix hackers struggled with this in many ways. In the languages of 1970 function calls were expensive, either because call semantics were complicated (PL/1. Algol) or because the compiler was optimizing for other things like fast inner loops at the expense of call time. Thus, code tended to be written in big lumps. Ken and several of the other early Unix developers knew modularity was a good idea, but they remembered PL/1 and were reluctant to write small functions lest performance go to hell.

> Dennis Ritchie encouraged modularity by telling all and sundry that function calls were really, really cheap in C. Everybody started writing small functions and modularizing. Years later we found out that function calls were still expensive on the PDP-11, and VAX code was often spending 50% of its time in the CALLS instruction. Dennis had lied to us! But it was too late; we were all hooked...
>
> —
>
> <author>SteveJohnson</author>

All programmers today, Unix natives or not, are taught to modularize at the subroutine level within programs. Some learn the art of doing this at the module or abstract-data-type level and call that 'good design'. The design-patterns movement is making a noble effort to push up a level from there and discover successful design abstractions that can be applied to organize the large-scale structure of programs.

Getting better at all these kinds of problem partitioning is a worthy goal, and many excellent treatments of them are available elsewhere. We shall not attempt to cover all the issues relating to modularity within programs in too much detail: first, because that is a subject for an entire volume (or several volumes) in itself; and second, because this is a book about the art of *Unix* programming.

What we will do here is examine more specifically what the Unix tradition teaches us about how to follow the Rule of Modularity. In this chapter, our examples will live within process units. Later, in Chapter 7, we'll examine the circumstances under which partitioning programs into multiple cooperating processes is a good idea, and discuss more specific techniques for doing that partitioning.

Encapsulation and Optimal Module Size

The first and most important quality of modular code is *encapsulation*. Well-encapsulated modules don't expose their internals to each other. They don't call into the middle of each others' implementations, and they don't promiscuously share global data. They communicate using application programming interfaces (APIs) — narrow, well-defined sets of procedure calls and data structures. This is what the Rule of Modularity is about.

The APIs between modules have a dual role. On the implementation level, they function as choke points between the modules, preventing the internals of each from leaking into its neighbors. On the design level, it is the APIs (not the bits of implementation between them) that really define your architecture.

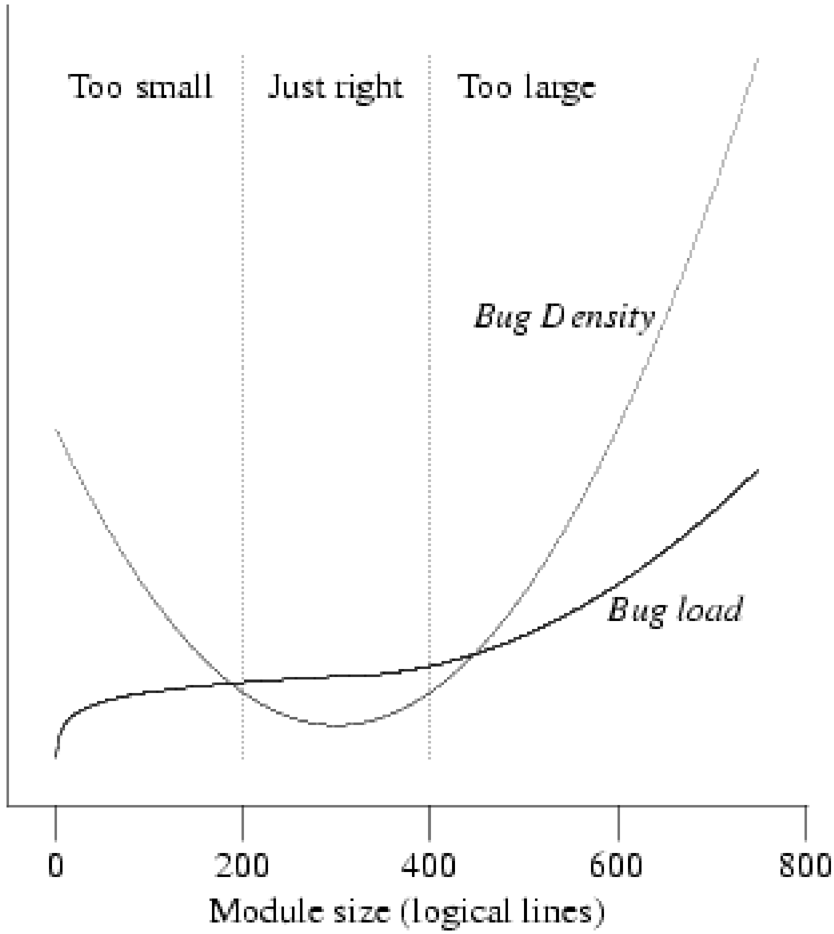
One good test for whether an API is well designed is this one: if you try to write a description of it in purely human language (with no source-code extracts allowed), does it make sense? It is a very good idea to get into the habit of writing informal descriptions of your APIs before you code them. Indeed, some of the most able developers start by defining their interfaces, writing brief comments to describe them, and then writing the code — since the process of writing the comment clarifies what the code must do. Such descriptions help you organize your thoughts, they make useful module comments, and eventually you might want to turn them into a roadmap document for future readers of the code.

As you push module decomposition harder, the pieces get smaller and the definition of the APIs gets more important. Global complexity, and consequent vulnerability to bugs, decreases. It has been received wisdom in computer science since the 1970s (exemplified in papers such as [Parnas]) that you ought to design your software systems as hierarchies of nested modules, with the grain size of the modules at each level held to a minimum.

It is possible, however, to push this kind of decomposition too hard and make your modules too small. There is evidence [Hatton97] that when one plots defect density versus module size, the curve is U-shaped and concave upwards (see Figure 4.1). Very small and very large modules are associated with more bugs than those of intermediate size. A different way of viewing the same data is to plot lines of code per module versus total bugs. The curve looks roughly logarithmic up to a 'sweet spot' where it flattens (corresponding to the minimum in the defect density curve), after which it goes up as the square of the number of the lines of code (which is what one might intuitively expect for the whole curve, following Brooks's Law[41]).

[41] Brooks's Law predicts that adding programmers to a late project makes it later. More generally, it predicts that costs and error rates rise as the square of the number of programmers on a project.

Figure 4.1. Qualitative plot of defect count and density vs. module size.



This unexpectedly increasing incidence of bugs at small module sizes holds across a wide variety of systems implemented in different languages. Hatton has proposed a model relating this nonlinearity to the chunk size of human short-term memory.[42] Another way to interpret the nonlinearity is that at

[42] In Hatton's model, small differences in the maximum chunk size a programmer can hold in short-term memory have a large multiplicative effect on the programmer's efficiency. This might be a major contributor to the order-of-magnitude (or larger) variations in effectiveness observed by Fred Brooks and others.

small module grain sizes, the increasing complexity of the interfaces becomes the dominating term; it's difficult to read the code because you have to understand everything before you can understand anything. In Chapter 7 we'll examine more advanced forms of program partitioning; there, too, the complexity of interface protocols comes to dominate the total complexity of the system as the component processes get smaller.

In nonmathematical terms, Hatton's empirical results imply a sweet spot between 200 and 400 logical lines of code that minimizes probable defect density, all other factors (such as programmer skill) being equal. This size is independent of the language being used — an observation which strongly reinforces the advice given elsewhere in this book to program with the most powerful languages and tools you can. Beware of taking these numbers too literally however. Methods for counting lines of code vary considerably according to what the analyst considers a logical line, and other biases (such as whether comments are stripped). Hatton himself suggests as a rule of thumb a 2x conversion between logical and physical lines, suggesting an optimal range of 400–800 physical lines.

Compactness and Orthogonality

Code is not the only sort of thing with an optimal chunk size. Languages and APIs (such as sets of library or system calls) run up against the same sorts of human cognitive constraints that produce Hatton's U-curve.

Accordingly, Unix programmers have learned to think very hard about two other properties when designing APIs, command sets, protocols, and other ways to make computers do tricks: *compactness* and *orthogonality*.

Compactness

Compactness is the property that a design can fit inside a human being's head. A good practical test for compactness is this: Does an experienced user normally need a manual? If not, then the design (or at least the subset of it that covers normal use) is compact.

Compact software tools have all the virtues of physical tools that fit well in the hand. They feel pleasant to use, they don't obtrude themselves between your mind and your work, they make you more productive — and they are much less likely than unwieldy tools to turn in your hand and injure you.

Compact is not equivalent to ‘weak’. A design can have a great deal of power and flexibility and still be compact if it is built on abstractions that are easy to think about and fit together well. Nor is compact equivalent to ‘easily learned’; some compact designs are quite difficult to understand until you have mastered an underlying conceptual model that is tricky, at which point your view of the world changes and compact *becomes* simple. For a lot of people, the Lisp language is a classic example of this.

> Nor does compact mean ‘small’. If a well-designed system is predictable and ‘obvious’ to the experienced user, it might have quite a few pieces.
>
> —
>
> <author>KenArnold</author>

Very few software designs are compact in an absolute sense, but many are compact in a slightly looser sense of the term. They have a compact working set, a subset of capabilities that suffices for 80% or more of what expert users normally do with them. Practically speaking, such designs normally need a reference card or cheat sheet but not a manual. We’ll call such designs *semi-compact*, as opposed to *strictly compact*.

The concept is perhaps best illustrated by examples. The Unix system call API is semi-compact, but the standard C library is not compact in any sense. While Unix programmers easily keep a subset of the system calls sufficient for most applications programming (file system operations, signals, and process control) in their heads, the C library on modern Unixes includes many hundreds of entry points, e.g., mathematical functions, that won’t all fit inside a single programmer’s cranium.

The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information [Miller] is one of the foundation papers in cognitive psychology (and, incidentally, the specific reason that U.S. local telephone numbers have seven digits). It showed that the number of discrete items of information human beings can hold in short-term memory is seven, plus or minus two. This gives us a good rule of thumb for evaluating the compactness of APIs: Does a programmer have to remember more than seven entry points? Anything larger than this is unlikely to be strictly compact.

Among Unix tools, make(1) is compact; autoconf(1) and automake(1) are not. Among markup languages, HTML is semi-compact, but DocBook (a documentation markup language we shall discuss in Chapter 18) is not. The man(7) macros are compact, but troff(1) markup is not.

Among general-purpose programming languages, C and Python are semi-compact; Perl, Java, Emacs Lisp, and shell are not (especially since serious shell programming requires you to know

half-a-dozen other tools like sed(1) and awk(1)). C++ is anti-compact — the language's designer has admitted that he doesn't expect any one programmer to ever understand it all.

Some designs that are not compact have enough internal redundancy of features that individual programmers end up carving out compact dialects sufficient for that 80% of common tasks by choosing a working subset of the language. Perl has this kind of pseudo-compactness, for example. Such designs have a built-in trap; when two programmers try to communicate about a project, they may find that differences in their working subsets are a significant barrier to understanding and modifying the code.

Noncompact designs are not automatically doomed or bad, however. Some problem domains are simply too complex for a compact design to span them. Sometimes it's necessary to trade away compactness for some other virtue, like raw power and range. Troff markup is a good example of this. So is the BSD sockets API. The purpose of emphasizing compactness as a virtue is not to condition you to treat compactness as an absolute requirement, but to teach you to do what Unix programmers do: value compactness properly, design for it whenever possible, and not throw it away casually.

Orthogonality

Orthogonality is one of the most important properties that can help make even complex designs compact. In a purely orthogonal design, operations do not have side effects; each action (whether it's an API call, a macro invocation, or a language operation) changes just one thing without affecting others. There is one and only one way to change each property of whatever system you are controlling.

Your monitor has orthogonal controls. You can change the brightness independently of the contrast level, and (if the monitor has one) the color balance control will be independent of both. Imagine how much more difficult it would be to adjust a monitor on which the brightness knob affected the color balance: you'd have to compensate by tweaking the color balance every time after you changed the brightness. Worse, imagine if the contrast control also affected the color balance; then, you'd have to adjust both knobs simultaneously in exactly the right way to change either contrast or color balance alone while holding the other constant.

Far too many software designs are non-orthogonal. One common class of design mistake, for example, occurs in code that reads and parses data from one (source) format to another (target) format. A designer who thinks of the source format as always being stored in a disk file may write the conversion function to open and read from a named file. Usually the input could just as well

have been any file handle. If the conversion routine were designed orthogonally, e.g., without the side effect of opening a file, it could save work later when the conversion has to be done on a data stream supplied from standard input, a network socket, or any other source.

Doug McIlroy's advice to "Do one thing well" is usually interpreted as being about simplicity. But it's also, implicitly and at least as importantly, about orthogonality.

It's not a problem for a program to do one thing well and other things as side effects, provided supporting those other things doesn't raise the complexity of the program and its vulnerability to bugs. In Chapter 9 we'll examine a program called *ascii* that prints synonyms for the names of ASCII characters, including hex, octal, and binary values; as a side effect, it can serve as a quick base converter for numbers in the range 0–255. This second use is not an orthogonality violation because the features that support it are all necessary to the primary function; they do not make the program more difficult to document or maintain.

The problems with non-orthogonality arise when side effects complicate a programmer's or user's mental model, and beg to be forgotten, with results ranging from inconvenient to dire. Even when you do not forget the side effects, you're often forced to do extra work to suppress them or work around them.

There is an excellent discussion of orthogonality and how to achieve it in *The Pragmatic Programmer* [Hunt-Thomas]. As they point out, orthogonality reduces test and development time, because it's easier to verify code that neither causes side effects nor depends on side effects from other code — there are fewer combinations to test. If it breaks, orthogonal code is more easily replaced without disturbance to the rest of the system. Finally, orthogonal code is easier to document and reuse.

The concept of *refactoring*, which first emerged as an explicit idea from the 'Extreme Programming' school, is closely related to orthogonality. To refactor code is to change its structure and organization without changing its observable behavior. Software engineers have been doing this since the birth of the field, of course, but naming the practice and identifying a stock set of refactoring techniques has helped concentrate peoples' thinking in useful ways. Because these fit so well with the central concerns of the Unix design tradition, Unix developers have quickly coopted the terminology and ideas of refactoring.[43]

[43] In the foundation text on this topic, *Refactoring* [Fowler], the author comes very close to stating that the principal goal of refactoring is to improve orthogonality. But lacking the concept, he can only approximate this idea from several different directions: eliminating code duplication and various other "bad smells" many of which are some sort of orthogonality violation.

The basic Unix APIs were designed for orthogonality with imperfect but considerable success. We take for granted being able to open a file for write access without exclusive-locking it for write, for example; not all operating systems are so graceful. Old-style (System III) signals were non-orthogonal, because signal receipt had the side-effect of resetting the signal handler to the default die-on-receipt. There are large non-orthogonal patches like the BSD sockets API and *very* large ones like the X windowing system's drawing libraries.

But on the whole the Unix API is a good example: Otherwise it not only would not but *could* not be so widely imitated by C libraries on other operating systems. This is also a reason that the Unix API repays study even if you are not a Unix programmer; it has lessons about orthogonality to teach.

The SPOT Rule

The Pragmatic Programmer articulates a rule for one particular kind of orthogonality that is especially important. Their "Don't Repeat Yourself" rule is: every piece of knowledge must have a *single*, unambiguous, authoritative representation within a system. In this book we prefer, following a suggestion by Brian Kernighan, to call this the Single Point Of Truth or SPOT rule.

Repetition leads to inconsistency and code that is subtly broken, because you changed only some repetitions when you needed to change *all* of them. Often, it also means that you haven't properly thought through the organization of your code.

Constants, tables, and metadata should be declared and initialized *once* and imported elsewhere. Any time you see duplicate code, that's a danger sign. Complexity is a cost; don't pay it twice.

Often it's possible to remove code duplication by *refactoring*; that is, changing the organization of your code without changing the core algorithms. Data duplication sometimes appears to be forced on you. But when you see it, here are some valuable questions to ask:

- If you have duplicated data in your code because it has to have two different representations in two different places, can you write a function, tool or code generator to make one representation from the other, or both from a common source?
- If your documentation duplicates knowledge in your code, can you generate parts of the documentation from parts of the code, or vice-versa, or both from a common higher-level representation?

- If your header files and interface declarations duplicate knowledge in your implementation code, is there a way you can generate the header files and interface declarations from the code?

There is an analog of the SPOT rule for data structures: “No junk, no confusion”. “No junk” says that the data structure (the model) should be minimal, e.g., not made so general that it can represent situations which cannot exist. “No confusion” says that states which must be kept distinct in the real-world problem must be kept distinct in the model. In short, the SPOT rule advocates seeking a data structure whose states have a one-to-one correspondence with the states of the real-world system to be modeled.

From deeper within the Unix tradition, we can add some of our own corollaries of the SPOT rule:

- Are you duplicating data because you’re caching intermediate results of some computation or lookup? Consider carefully whether this is premature optimization; stale caches (and the layers of code needed to keep caches synchronized) are a fertile source of bugs,[44] and can even slow down overall performance if (as often happens) the cache-management overhead is higher than you expected.
- If you see lots of duplicative boilerplate code, can you generate all of it from a single higher-level representation, twiddling a few knobs to generate the different cases?

The reader should begin to see a pattern emerging here.

In the Unix world, the SPOT Rule as a unifying idea has seldom been explicit — but heavy use of code generators to implement particular *kinds* of SPOT are very much part of the tradition. We’ll survey these techniques in Chapter 9.

Compactness and the Strong Single Center

One subtle but powerful way to promote compactness in a design is to organize it around a strong core algorithm addressing a clear formal definition of the problem, avoiding heuristics and fudging.

> Formalization often clarifies a task spectacularly. It is not enough for a programmer to recognize that bits of his task fall within standard computer-science categories — a little depth-first search here and a quicksort there. The

[44] An archetypal example of bad caching is the **rehash** directive in csh(1); type **man 1 csh** for details. See the section called “Caching Operation Results” for another example.

> best results occur when the nub of the task can be formalized, and a clear model of the job at hand can be constructed. It is not necessary that ultimate users comprehend the model. The very existence of a unifying core will provide a comfortable feel, unencumbered with the why-in-hell-did-they-do-that moments that are so prevalent in using Swiss-army-knife programs.
>
> —
>
> <author>DougMcIlroy</author>

This is an often-overlooked strength of the Unix tradition. Many of its most effective tools are thin wrappers around a direct translation of some single powerful algorithm.

Perhaps the clearest example of this is diff(1), the Unix tool for reporting differences between related files. This tool and its dual, patch(1), have become central to the network-distributed development style of modern Unix. A valuable property of diff is that it seldom surprises anyone. It doesn't have special cases or painful edge conditions, because it uses a simple, mathematically sound method of sequence comparison. This has consequences:

> By virtue of a mathematical model and a solid algorithm, Unix diff contrasts markedly with its imitators. First, the central engine is solid, small, and has never needed one line of maintenance. Second, the results are clear and consistent, unmarred by surprises where heuristics fail.
>
> —
>
> <author>DougMcIlroy</author>

Thus, people who use diff can develop an intuitive feel for what it will do in any given situation without necessarily understanding the central algorithm perfectly. Other well-known examples of this special kind of clarity achieved through a strong central algorithm abound in Unix:

- The grep(1) utility for selecting lines out of files by pattern matching is a simple wrapper around a formal algebra of regular-expression patterns (see the section called "Case Study: Regular Expressions" for discussion). If it had lacked this consistent mathematical model, it would probably look like the design of the original glob(1) facility in the oldest Unixes, a handful of ad-hoc wildcards that can't be combined.

- The yacc(1) utility for generating language parsers is a thin wrapper around the formal theory of LR(1) grammars. Its partner, the lexical analyzer generator lex(1), is a similarly thin wrapper around the theory of nondeterministic finite-state automata.

All three of these programs are so bug-free that their correct functioning is taken utterly for granted, and compact enough to fit easily in a programmer's hand. Only a part of these good qualities are due to the polishing that comes with a long service life and frequent use; most of it is that, having been constructed around a strong and provably correct algorithmic core, they never needed much polishing in the first place.

The opposite of a formal approach is using *heuristics*—rules of thumb leading toward a solution that is probabilistically, but not certainly, correct. Sometimes we use heuristics because a deterministically correct solution is impossible. Think of spam filtering, for example; an algorithmically perfect spam filter would need a full solution to the problem of understanding natural language as a module. Other times, we use heuristics because known formally correct methods are impossibly expensive. Virtual-memory management is an example of this; there are near-perfect solutions, but they require so much runtime instrumentation that their overhead would swamp any theoretical gain over heuristics.

The trouble with heuristics is that they proliferate special cases and edge cases. If nothing else, you usually have to backstop a heuristic with some sort of recovery mechanism when it fails. All the usual problems with escalating complexity follow. To manage the resulting tradeoffs, you have to start by being aware of them. Always ask if a heuristic actually pays off in performance what it costs in code complexity — and don't guess at the performance difference, actually measure it before making a decision.

The Value of Detachment

We began this book with a reference to Zen: "a special transmission, outside the scriptures". This was not mere exoticism for stylistic effect; the core concepts of Unix have always had a spare, Zen-like simplicity that continues to shine through the layers of historical accidents that have accreted around them. This quality is reflected in the cornerstone documents of Unix, like *The C Programming Language* [Kernighan-Ritchie] and the 1974 CACM paper that introduced Unix to the world; one of the famous quotes from that paper observes "...constraint has encouraged not only economy, but also a certain elegance of design". That simplicity came from trying to think not about how much a language or operating system could do, but of how *little* it could do — not by carrying assumptions but by starting from zero (what in Zen is called "beginner's mind" or "empty mind").

To design for compactness and orthogonality, start from zero. Zen teaches that attachment leads to suffering; experience with software design teaches that attachment to unnoticed assumptions leads to non-orthogonality, noncompact designs, and projects that fail or become maintenance nightmares.

To achieve enlightenment and surcease from suffering, Zen teaches detachment. The Unix tradition teaches the value of detachment from the particular, accidental conditions under which a design problem was posed. Abstract. Simplify. Generalize. Because we write software to solve problems, we cannot completely detach from the problems — but it is well worth the mental effort to see how many preconceptions you can throw away, and whether the design becomes more compact and orthogonal as you do that. Possibilities for code reuse often result.

Jokes about the relationship between Unix and Zen are a live part of the Unix tradition as well.[45] This is not an accident.

Software Is a Many-Layered Thing

Broadly speaking, there are two directions one can go in designing a hierarchy of functions or objects. Which direction you choose, and when, has a profound effect on the layering of your code.

Top-Down versus Bottom-Up

One direction is bottom-up, from concrete to abstract — working up from the specific operations in the problem domain that you know you will need to perform. For example, if one is designing firmware for a disk drive, some of the bottom-level primitives might be ‘seek head to physical block’, ‘read physical block’, ‘write physical block’, ‘toggle drive LED’, etc.

The other direction is top-down, abstract to concrete — from the highest-level specification describing the project as a whole, or the application logic, downwards to individual operations. Thus, if one is designing software for a mass-storage controller that might drive several different sorts of media, one might start with abstract operations like ‘seek logical block’, ‘read logical block’, ‘write logical block’, ‘toggle activity indication’. These would differ from the similarly named hardware-level operations above in that they’re intended to be generic across different kinds of physical devices.

These two examples could be two ways of approaching design for the same collection of hardware. Your choice, in cases like this, is one of these: either abstract the hardware (so the objects encapsulate the real things out there and the program is merely a list of manipulations on those things), or organize around some behavioral model (and then embed the actual hardware manipulations that carry it out in the flow of the behavioral logic).

[45] For a recent example of Unix/Zen crossover, see Appendix D.

An analogous choice shows up in a lot of different contexts. Suppose you're writing MIDI sequencer software. You could organize that code around its top level (sequencing tracks) or around its bottom level (switching patches or samples and driving wave generators).

A very concrete way to think about this difference is to ask whether the design is organized around its main event loop (which tends to have the high-level application logic close to it) or around a service library of all the operations that the main loop can invoke. A designer working from the top down will start by thinking about the program's main event loop, and plug in specific events later. A designer working from the bottom up will start by thinking about encapsulating specific tasks and glue them together into some kind of coherent order later on.

For a larger example, consider the design of a Web browser. The top-level design of a Web browser is a specification of the expected behavior of the browser: what types of URL (like `http:` or `ftp:` or `file:`) it interprets, what kinds of images it is expected to be able to render, whether and with what limitations it will accept Java or JavaScript, etc. The layer of the implementation that corresponds to this top-level view is its main event loop; each time around, the loop waits for, collects, and dispatches on a user action (such as clicking a Web link or typing a character into a field).

But the Web browser has to call a large set of domain primitives to do its job. One group of these is concerned with establishing network connections, sending data over them, and receiving responses. Another set is the operations of the GUI toolkit the browser will use. Yet a third set might be concerned with the mechanics of parsing retrieved HTML from text into a document object tree.

Which end of the stack you start with matters a lot, because the layer at the other end is quite likely to be constrained by your initial choices. In particular, if you program purely from the top down, you may find yourself in the uncomfortable position that the domain primitives your application logic wants don't match the ones you can actually implement. On the other hand, if you program purely from the bottom up, you may find yourself doing a lot of work that is irrelevant to the application logic — or merely designing a pile of bricks when you were trying to build a house.

Ever since the structured-programming controversies of the 1960s, novice programmers have generally been taught that the correct approach is the top-down one: stepwise refinement, where you specify what your program is to do at an abstract level and gradually fill in the blanks of implementation until you have concrete working code. Top-down tends to be good practice when three preconditions are true: (a) you can specify in advance precisely what the program is to do, (b) the specification is unlikely to change significantly during implementation, and (c) you have a lot of freedom in choosing, at a low level, how the program is to get that job done.

These conditions tend to be fulfilled most often in programs relatively close to the user and high in the software stack — applications programming. But even there those preconditions often fail. You can't count on knowing what the 'right' way for a word processor or a drawing program to behave is until the user interface has had end-user testing. Purely top-down programming often has the effect of overinvesting effort in code that has to be scrapped and rebuilt because the interface doesn't pass a reality check.

In self-defense against this, programmers try to do both things — express the abstract specification as top-down application logic, *and* capture a lot of low-level domain primitives in functions or libraries, so they can be reused when the high-level design changes.

Unix programmers inherit a tradition that is centered in systems programming, where the low-level primitives are hardware-level operations that are fixed in character and extremely important. They therefore lean, by learned instinct, more toward bottom-up programming.

Whether you're a systems programmer or not, bottom-up can also look more attractive when you are programming in an exploratory way, trying to get a grasp on hardware or software or real-world phenomena you don't yet completely understand. Bottom-up programming gives you time and room to refine a vague specification. Bottom-up also appeals to programmers' natural human laziness — when you have to scrap and rebuild code, you tend to have to throw away larger pieces if you're working top-down than you do if you're working bottom-up.

Real code, therefore tends to be programmed both top-down and bottom-up. Often, top-down and bottom-up code will be part of the same project. That's where 'glue' enters the picture.

Glue Layers

When the top-down and bottom-up drives collide, the result is often a mess. The top layer of application logic and the bottom layer of domain primitives have to be impedance-matched by a layer of glue logic.

One of the lessons Unix programmers have learned over decades is that glue is nasty stuff and that it is vitally important to keep glue layers as thin as possible. Glue should stick things together, but should not be used to hide cracks and unevenness in the layers.

In the Web-browser example, the glue would include the rendering code that maps a document object parsed from incoming HTML into a flattened visual representation as a bitmap in a display buffer, using GUI domain primitives to do the painting. This rendering code is notoriously the most

bug-prone part of a browser. It attracts into itself kluges to address problems that originate both in the HTML parsing (because there is a lot of ill-formed markup out there) and the GUI toolkit (which may not have quite the primitives that are really needed).

A Web browser's glue layer has to mediate not merely between specification and domain primitives, but between several different external specifications: the network behavior standardized in HTTP, HTML document structure, and various graphics and multimedia formats as well as the users' behavioral expectations from the GUI.

And one single bug-prone glue layer is not the worst fate that can befall a design. A designer who is aware that the glue layer exists, and tries to organize it into a middle layer around its own set of data structures or objects, can end up with *two* layers of glue — one above the midlayer and one below. Programmers who are bright but unseasoned are particularly apt to fall into this trap; they'll get each fundamental set of classes (application logic, midlayer, and domain primitives) right and make them look like the textbook examples, only to flounder as the multiple layers of glue needed to integrate all that pretty code get thicker and thicker.

The thin-glue principle can be viewed as a refinement of the Rule of Separation. Policy (the application logic) should be cleanly separated from mechanism (the domain primitives), but if there is a lot of code that is neither policy nor mechanism, chances are that it is accomplishing very little besides adding global complexity to the system.

Case Study: C Considered as Thin Glue

The C language itself is a good example of the effectiveness of thin glue.

In the late 1990s, Gerrit Blaauw and Fred Brooks observed in *Computer Architecture: Concepts and Evolution* [BlaauwBrooks] that the architectures in every generation of computers, from early mainframes through minicomputers through workstations through PCs, had tended to converge. The later a design was in its technology generation, the more closely it approximated what Blaauw & Brooks called the "classical architecture": binary representation, flat address space, a distinction between memory and working store (registers), general-purpose registers, address resolution to fixed-length bytes, two-address instructions, big-endianness,[46] and data types a consistent set with sizes a multiple of either 4 or 6 bits (the 6-bit families are now extinct).

[46]The terms *big-endian* and *little-endian* refer to architectural choices about the order in which bits are interpreted within a machine word. Though it has no canonical location, a Web search for *On Holy Wars and a Plea for Peace* should turn up a classic and entertaining paper on this subject.

Thompson and Ritchie designed C to be a sort of structured assembler for an idealized processor and memory architecture that they expected could be efficiently modeled on most conventional computers. By happy accident, their model for the idealized processor was the PDP-11, a particularly mature and elegant minicomputer design that closely approximated Blaauw & Brooks's classical architecture. By good judgment, Thompson and Ritchie declined to wire into their language most of the few traits (such as little-endian byte order) where the PDP-11 didn't match it.[47]

The PDP-11 became an important model for the following generations of microprocessor architectures. The basic abstractions of C turned out to capture the classical architecture rather neatly. Thus, C started out as a good fit for microprocessors and, rather than becoming irrelevant as its assumptions fell out of date, actually became a *better* fit as hardware converged more closely on the classical architecture. One notable example of this convergence was when Intel's 386, with its large flat memory-address space, replaced the 286's awkward segmented-memory addressing after 1985; pure C was actually a better fit for the 386 than it had been for the 286.

It is not a coincidence that the experimental era in computer architectures ended in the mid-1980s at the same time that C (and its close descendant C++) were sweeping all before them as general-purpose programming languages. C, designed as a thin but flexible layer over the classical architecture, looks with two decades' additional perspective like almost the best possible design for the structured-assembler niche it was intended to fill. In addition to compactness, orthogonality, and detachment (from the machine architecture on which it was originally designed), it also has the important quality of transparency that we will discuss in Chapter 6. The few language designs since that are arguably better have needed to make large changes (like introducing garbage collection) in order to get enough functional distance from C not to be swamped by it.

This history is worth recalling and understanding because C shows us how powerful a clean, minimalist design can be. If Thompson and Ritchie had been less wise, they would have designed a language that did much more, relied on stronger assumptions, never ported satisfactorily off its original hardware platform, and withered away as the world changed out from under it. Instead, C has flourished — and the example Thompson and Ritchie set has influenced the style of Unix development ever since. As the writer, adventurer, artist, and aeronautical engineer Antoine de Saint-Exupéry once put it, writing about the design of airplanes: *«La perfection est atteinte non quand il ne reste rien à ajouter, mais quand il ne reste rien à enlever»*. ("Perfection is attained not when there is nothing more to add, but when there is nothing more to remove".)

[47]The widespread belief that the autoincrement and autodecrement features entered C because they represented PDP-11 machine instructions is a myth. According to Dennis Ritchie, these operations were present in the ancestral B language before the PDP-11 existed.

Ritchie and Thompson lived by this maxim. Long after the resource constraints on early Unix software had eased, they worked at keeping C as thin a layer over the hardware as possible.

> Dennis used to say to me, when I would ask for some particularly extravagant feature in C, “If you want PL/1, you know where to get it”. He didn’t have to deal with some marketer saying “But we need a check in the box on the sales viewgraph!”
>
> —
>
> <author>MikeLesk</author>

The history of C is also a lesson in the value of having a working reference implementation *before* you standardize. We’ll return to this point in Chapter 17 when we discuss the evolution of C and Unix standards.

Libraries

One consequence of the emphasis that the Unix programming style put on modularity and well-defined APIs is a strong tendency to factor programs into bits of glue connecting collections of libraries, especially shared libraries (the equivalents of what are called dynamically-linked libraries or DLLs under Windows and other operating systems).

If you are careful and clever about design, it is often possible to partition a program so that it consists of a user-interface-handling main section (policy) and a collection of service routines (mechanism) with effectively no glue at all. This approach is especially appropriate when the program has to do a lot of very specific manipulations of data structures like graphic images, network-protocol packets, or control blocks for a hardware interface. Some good general architectural advice from within the Unix tradition, particularly applicable to the resource-management challenges of this sort of library is collected in *The Discipline and Method Architecture for Reusable Libraries* [Vo].

Under Unix, it is normal practice to make this layering explicit, with the service routines collected in a library that is separately documented. In such programs, the front end gets to specialize in user-interface considerations and high-level protocol. With a little more care in design, it may be possible to detach the original front end and replace it with others adapted for different purposes. Some other advantages should become evident from our case study.

There is a flip side to this. In the Unix world, libraries which are delivered *as libraries* should come with exerciser programs.

> APIs should come with programs, and vice versa. An API that you must write C code to use, which cannot be invoked easily from the command line, is harder to learn and use. And contrariwise, it's a royal pain to have interfaces whose *only* open, documented form is a program, so you cannot invoke them easily from a C program — for example, route(1) in older Linuxes.
>
> —
>
> <author>HenrySpencer</author>

Besides easing the learning curve, library exercisers often make excellent test frameworks. Experienced Unix programmers therefore see them not just as a form of thoughtfulness to the library's users but as an indication that the code has probably been well tested.

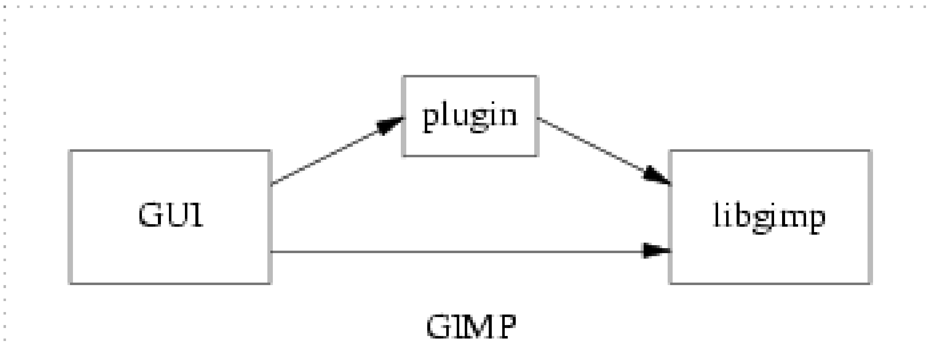
An important form of library layering is the *plugin*, a library with a set of known entry points that is dynamically loaded after startup time to perform a specialized task. For plugins to work, the calling program has to be organized largely as a documented service library that the plugin can call back into.

Case Study: GIMP Plugins

The GIMP (GNU Image Manipulation program) is a graphics editor designed to be driven through an interactive GUI. But GIMP is built as a library of image-manipulation and housekeeping routines called by a relatively thin layer of control code. The driver code knows about the GUI, but not directly about image formats; the library routines reverse this by knowing about image formats and operations but not about the GUI.

The library layer is documented (and, in fact shipped as "libgimp" for use by other programs). This means that C programs called "plugins" can be dynamically loaded by GIMP and call the library to do image manipulation, effectively taking over control at the same level as the GUI (see Figure 4.2).

Figure 4.2. Caller/callee relationships in GIMP with a plugin loaded.



Plugins are used to perform lots of special-purpose transformations such as colormap hacking, blurring and despeckling; also for reading and writing file formats not native to the GIMP core; for extensions like editing animations and window manager themes; and for lots of other sorts of image-hacking that can be automated by scripting the image-hacking logic in the GIMP core. A registry of GIMP plugins is available on the World Wide Web.

Though most GIMP plugins are small, simple C programs, it is also possible to write a plugin that exposes the library API to a scripting language; we'll discuss this possibility in Chapter 11 when we examine the 'polyvalent program' pattern.

Unix and Object-Oriented Languages

Since the mid-1980s most new language designs have included native support for *object-oriented programming* (OO). Recall that in object-oriented programming, the functions that act on a particular data structure are encapsulated with the data in an object that can be treated as a unit. By contrast, modules in non-OO languages make the association between data and the functions that act on it rather accidental, and modules frequently leak data or bits of their internals into each other.

The OO design concept initially proved valuable in the design of graphics systems, graphical user interfaces, and certain kinds of simulation. To the surprise and gradual disillusionment of many, it has proven difficult to demonstrate significant benefits of OO outside those areas. It's worth trying to understand why.

There is some tension and conflict between the Unix tradition of modularity and the usage patterns that have developed around OO languages. Unix programmers have always tended to be a bit more skeptical about OO than their counterparts elsewhere. Part of this is because of the Rule of Diversity; OO has far too often been promoted as the One True Solution to the software-complexity problem. But there is something else behind it as well, an issue which is worth exploring as background before we evaluate specific OO (object-oriented) languages in Chapter 14. It will also help throw some characteristics of the Unix style of non-OO programming into sharper relief.

We observed above that the Unix tradition of modularity is one of thin glue, a minimalist approach with few layers of abstraction between the hardware and the top-level objects of a program. Part of this is the influence of C. It takes serious effort to simulate true objects in C. Because that's so, piling up abstraction layers is an exhausting thing to do. Thus, object hierarchies in C tend to be relatively flat and transparent. Even when Unix programmers use other languages, they tend to want to carry over the thin-glue/shallow-layering style that Unix models have taught them.

OO languages make abstraction easy — perhaps too easy. They encourage architectures with thick glue and elaborate layers. This can be good when the problem domain is truly complex and demands a lot of abstraction, but it can backfire badly if coders end up doing simple things in complex ways just because they can.

All OO languages show some tendency to suck programmers into the trap of excessive layering. Object frameworks and object browsers are not a substitute for good design or documentation, but they often get treated as one. Too many layers destroy transparency: It becomes too difficult to see down through them and mentally model what the code is actually doing. The Rules of Simplicity, Clarity, and Transparency get violated wholesale, and the result is code full of obscure bugs and continuing maintenance problems.

This tendency is probably exacerbated because a lot of programming courses teach thick layering as a way to satisfy the Rule of Representation. In this view, having lots of classes is equated with embedding knowledge in your data. The problem with this is that too often, the 'smart data' in the glue layers is not actually about any natural entity in whatever the program is manipulating — it's just about being glue. (One sure sign of this is a proliferation of abstract subclasses or 'mixins'.)

Another side effect of OO abstraction is that opportunities for optimization tend to disappear. For example, $a + a + a + a$ can become $a * 4$ and even $a << 2$ if a is an integer. But if one creates a class with operators, there is nothing to indicate if they are commutative, distributive, or associative. Since one isn't supposed to look inside the object, it's not possible to know which of two equivalent expressions is more efficient. This isn't in itself a good reason to avoid using OO techniques on new

projects; that would be premature optimization. But it is reason to think twice before transforming non-OO code into a class hierarchy.

Unix programmers tend to share an instinctive sense of these problems. This tendency appears to be one of the reasons that, under Unix, OO languages have failed to displace non-OO workhorses like C, Perl (which actually has OO facilities, but they're not heavily used), and shell. There is more vocal criticism of OO in the Unix world than orthodoxy permits elsewhere; Unix programmers know when *not* to use OO; and when they do use OO languages, they spend more effort on trying to keep their object designs uncluttered. As the author of *The Elements of Networking Style* once observed in a slightly different context [Padlipsky]: "If you know what you're doing, three layers is enough; if you don't, even seventeen levels won't help".

One reason that OO has succeeded most where it has (GUIs, simulation, graphics) may be because it's relatively difficult to get the ontology of types wrong in those domains. In GUIs and graphics, for example, there is generally a rather natural mapping between manipulable visual objects and classes. If you find yourself proliferating classes that have no obvious mapping to what goes on in the display, it is correspondingly easy to notice that the glue has gotten too thick.

One of the central challenges of design in the Unix style is how to combine the virtue of detachment (simplifying and generalizing problems from their original context) with the virtue of thin glue and shallow, flat, transparent hierarchies of code and design.

We'll return to some of these points and apply them when we discuss object-oriented languages in Chapter 14.

Coding for Modularity

Modularity is expressed in good code, but it primarily comes from good design. Here are some questions to ask about any code you work on that might help you improve its modularity:

- How many global variables does it have? Global variables are modularity poison, an easy way for components to leak information to each other in careless and promiscuous ways.[48]

[48] Globals also mean your code cannot be reentrant; that is, multiple instances in the same process are likely to step on each other.

- Is the size of your individual modules in Hatton's sweet spot? If your answer is “No, many are larger”, you may have a long-term maintenance problem. Do you know what your own sweet spot is? Do you know what it is for other programmers you are cooperating with? If not, best be conservative and stick to sizes near the low end of Hatton's range.

- Are the individual functions in your modules too large? This is not so much a matter of line count as it is of internal complexity. If you can't informally describe a function's contract with its callers in one line, the function is probably too large.[49]

  Personally I tend to break up a subprogram when there are too many local variables. Another clue is [too many] levels of indentation. I rarely look at length.
  —
  <author>KenThompson</author>

- Does your code have internal APIs — that is, collections of function calls and data structures that you can describe to others as units, each sealing off some layer of function from the rest of the code? A good API makes sense and is understandable without looking at the implementation behind it. The classic test is this: Try to describe it to another programmer over the phone. If you fail, it is very probably too complex, and poorly designed.

- Do any of your APIs have more than seven entry points? Do any of your classes have more than seven methods each? Do your data structures have more than seven members?

- What is the distribution of the number of entry points per module across the project?[50] Does it seem uneven? Do the modules with lots of entry points really need that many? Module complexity tends to rise as the square of the number of entry points, too — yet another reason simple APIs are better than complicated ones.

You might find it instructive to compare these with our checklist of questions about transparency, and discoverability in Chapter 6.

[49]Many years ago, I learned from Kernighan & Plauger's *The Elements of Programming Style* a useful rule. Write that one-line comment immediately after the prototype of your function. For *every* function, without exception.
[50]A cheap way to collect this information is to analyze the tags files generated by a utility like etags(1) or ctags(1).

Chapter 5. Textuality

Good Protocols Make Good Practice

It's a well-known fact that computing devices such as the abacus were invented thousands of years ago. But it's not well known that the first use of a common computer protocol occurred in the Old Testament. This, of course, was when Moses aborted the Egyptians' process with a control-sea.

--

<author>TomGalloway</author>

`rec.arts.comics`, February 1992

In this chapter, we'll look at what the Unix tradition has to tell us about two different kinds of design that are closely related: the design of file formats for retaining application data in permanent storage, and the design of application protocols for passing data and commands between cooperating programs, possibly over a network.

What unifies these two kinds of design is that they both involve the serialization of in-memory data structures. For the internal operation of computer programs, the most convenient representation of a complex data structure is one in which all fields have the machine's native data format (e.g. two's-complement binary for integers) and all pointers are actual memory addresses (as opposed, say, to being named references). But these representations are not well suited to storage and transmission; memory addresses in the data structure lose their meaning outside memory, and emitting raw native data formats causes interoperability problems passing data between machines with different conventions (big- vs. little-endian, say, or 32-bit vs. 64-bit).

For transmission and storage, the traversable, quasi-spatial layout of data structures like linked lists needs to be flattened or serialized into a byte-stream representation from which the structure can later be recovered. The serialization (save) operation is sometimes called *marshaling* and its inverse (load) operation *unmarshaling*. These terms are usually applied with respect to objects in an OO language like C++ or Python or Java, but could be used with equal justice of operations like loading a graphics file into the internal storage of a graphics editor and saving it out after modifications.

A significant percentage of what C and C++ programmers maintain is ad-hoc code for marshaling and unmarshaling operations — even when the serialized representation chosen is as simple as a binary structure dump (a common technique under non-Unix environments). Modern languages like Python and Java tend to have built-in unmarshal and marshal functions that can be applied to any object or byte-stream representing an object, and that reduce this labor substantially.

But these naïve methods are often unsatisfactory for various reasons, including both the machine-interoperability problems we mentioned above and the negative trait of being opaque to other tools. When the application is a network protocol, economy may demand that an internal data structure (such as, say, a message with source and destination addresses) be serialized not into a single blob of data but into a series of attempted transactions or messages which the receiving machine may reject (so that, for example, a large message can be rejected if the destination address is invalid).

Interoperability, transparency, extensibility, and storage or transaction economy: these are the important themes in designing file formats and application protocols. Interoperability and transparency demand that we focus such designs on clean data representations, rather than putting convenience of implementation or highest possible performance first. Extensibility also favors textual protocols, since binary ones are often harder to extend or subset cleanly. Transaction economy sometimes pushes in the opposite direction — but we shall see that putting that criterion first is a form of premature optimization that it is often wise to resist.

Finally, we must note a difference between data file formats and the run-control files that are often used to set the startup options of Unix programs. The most basic difference is that (with sporadic exceptions like GNU Emacs's configuration interface) programs don't normally modify their own run-control files — the information flow is one-way, from file read at startup time to application settings. Data-file formats, on the other hand, associate properties with named resources and are both read and written by their applications. Configuration files are generally hand-edited and small, whereas data files are program-generated and can become arbitrarily large.

Historically, Unix has related but different sets of conventions for these two kinds of representation. The conventions for run control files are surveyed in Chapter 10; only conventions for data files are examined in this chapter.

The Importance of Being Textual

Pipes and sockets will pass binary data as well as text. But there are good reasons the examples we'll see in Chapter 7 are textual: reasons that hark back to Doug McIlroy's advice quoted in Chapter 1. Text streams are a valuable universal format because they're easy for human beings to read, write, and edit without specialized tools. These formats are (or can be designed to be) transparent.

Also, the very limitations of text streams help enforce encapsulation. By discouraging elaborate representations with rich, densely encoded structure, text streams also discourage programs from being promiscuous with each other about their internal states and help enforce encapsulation. We'll return to this point at the end of Chapter 7 when we discuss RPC.

When you feel the urge to design a complex binary file format, or a complex binary application protocol, it is generally wise to lie down until the feeling passes. If performance is what you're worried about, implementing compression on the text protocol stream either at some level below or above the application protocol will give you a cleaner and perhaps better-performing design than a binary protocol (text compresses well, and quickly).

> A bad example of binary formats in Unix history was the way device-independent *troff* read a binary file containing device information, supposedly for speed. The initial implementation generated that binary file from a text description in a somewhat unportable way. Faced with a need to port *ditroff quickly* to a new machine, rather than reinvent the binary goo, I ripped it out and just had *ditroff* read the text file. With carefully crafted file-reading code, the speed penalty was negligible.
>
> —
>
> <author>HenrySpencer</author>

Designing a textual protocol tends to future-proof your system. One specific reason is that ranges on numeric fields aren't implied by the format itself. Binary formats usually specify the number of bits allocated to a given value, and extending them is difficult. For example, IPv4 only allows 32 bits for an address. To extend address size to 128 bits (as done by IPv6) requires a major revamping.[51] In contrast, if you need a larger value in a text format, just write it. It may be that a given program can't receive values in that range, but it's usually easier to modify the program than to modify all the data stored in that format.

The only good justification for a binary protocol is if you're going to be manipulating large enough data sets that you're genuinely worried about getting the most bit-density out of your media, or if you're very concerned about the time or instruction budget required to interpret the data into an in-core structure. Formats for large images and multimedia are sometimes an example of the former, and network protocols with hard latency requirements sometimes an example of the latter.

> The reciprocal problem with SMTP or HTTP-like text protocols is that they tend to be expensive in bandwidth and slow to parse. The smallest X request is 4 bytes: the smallest HTTP request is about 100 bytes. X requests, including amortized overhead of transport, can be executed in the order of 100 instructions; at one point, an Apache [web server] developer proudly indicated they were down

[51] There is a legend that some early airline reservation systems allocated exactly one byte for a plane's passenger count. Supposedly they became very confused by the arrival of the Boeing 747, the first plane that could carry more than 255 passengers.

> to 7000 instructions. For graphics, bandwidth becomes everything on output; hardware is designed such that these days the graphics-card bus is *the* bottleneck for small operations, so any protocol had better be very tight if it is not to be a worse bottleneck. This is the extreme case.
>
> —
> <author>JimGettys</author>

These concerns are valid in other extreme cases as well as in X — for example, in the design of graphics file formats intended to hold very large images. But they are usually just another case of premature-optimization fever. Textual formats don't necessarily have much lower bit density than binary ones; they do after all use seven out of eight bits per byte. And what you gain by not having to parse text, you generally lose the first time you need to generate a test load, or to eyeball a program-generated example of your format and figure out what's in there.

In addition, the kind of thinking that goes into designing tight binary formats tends to fall down on making them cleanly extensible. The X designers experienced this:

> Against the current X framework is the fact we didn't design enough of a structure to make it easier to ignore trivial extensions to the protocol; we can do this some of the time, but a bit better framework would have been good.
>
> —
> <author>JimGettys</author>

When you think you have an extreme case that justifies a binary file format or protocol, you need to think very carefully about extensibility and leaving room in the design for growth.

Case Study: Unix Password File Format

On many operating systems, the per-user data required to validate logins and start a user's session is an opaque binary database. Under Unix, by contrast, it's a text file with records one per line and colon-separated fields.

Example 5.1 consists of some randomly-chosen example lines:

Example 5.1. Password file example.

```
games:*:12:100:games:/usr/games:
gopher:*:13:30:gopher:/usr/lib/gopher-data:
```

```
ftp:*:14:50:FTP User:/home/ftp:
esr:0SmFuPnH5JlNs:23:23:Eric S. Raymond:/home/esr:
nobody:*:99:99:Nobody:/:
```

Without even knowing anything about the semantics of the fields, we can notice that it would be hard to pack the data much tighter in a binary format. The colon sentinel characters would have to have functional equivalents taking at least as much space (usually either count bytes or NULs). The per-user records would either have to have terminators (which could hardly be shorter than a single newline) or else be wastefully padded out to a fixed length.

Actually the prospects for saving space through binary encoding pretty much vanish if you know the actual semantics of the data. The numeric user ID (3rd) and group ID (4th) fields are integers, thus on most machines a binary representation would be at least 4 bytes, and longer than the text for values up to 999. But let's agree to ignore this for now and suppose the best case that the numeric fields have a 0-255 range.

We could tighten up the numeric fields (3rd and 4th) by collapsing the numerics to single bytes, and the password strings (2nd) to an 8-bit encoding. On this example, that would give about an 8% size decrease.

That 8% of putative inefficiency buys us a lot. It avoids putting an arbitrary limit on the range of the numeric fields. It gives us the ability to modify the password file with any old text editor of our choice, rather than having to build a specialized tool to edit a binary format (though in the case of the password file itself, we have to be extra careful about concurrent edits). And it gives us the ability to do ad-hoc searches and filters and reports on the user account information with text-stream tools such as grep(1).

We do have to be a bit careful about not embedding a colon in any of the textual fields. Good practice is to tell the file write code to precede embedded colons with an escape character, and then to tell the file read code to interpret it. Unix tradition favors backslash for this use.

The fact that structural information is conveyed by field position rather than an explicit tag makes this format faster to read and write, but a bit rigid. If the set of properties associated with a key is expected to change with any frequency, one of the tagged formats described below might be a better choice.

Economy is not a major issue with password files to begin with, as they're normally read seldom[52] and infrequently modified. Interoperability is not an issue, since various data in the file (notably user and group numbers) are not portable off the originating machine. For password files, it's therefore quite clear that going where the transparency criterion leads was the right thing.

Case Study: `.newsrc` Format

Usenet news is a worldwide distributed bulletin-board system that anticipated today's P2P networking by two decades. It uses a message format very similar to that of RFC 822 electronic-mail messages, except that instead of being directed to personal recipients messages are sent to topic groups. Articles posted at any participating site are broadcast to each site that it has registered as a neighbor, and eventually flood-fill to all news sites.

Almost all Usenet news readers understand the `.newsrc` file, which records which Usenet messages have been seen by the calling user. Though it is named like a run-control file, it is not only read at startup but typically updated at the end of the newsreader run. The `.newsrc` format has been fixed since the first newsreaders around 1980. Example 5.2 is a representative section from a `.newsrc` file.

Example 5.2. A `.newsrc` example.

```
rec.arts.sf.misc! 1-14774,14786,14789
rec.arts.sf.reviews! 1-2534
rec.arts.sf.written: 1-876513
news.answers! 1-199359,213516,215735
news.announce.newusers! 1-4399
news.newusers.questions! 1-645661
news.groups.questions! 1-32676
news.software.readers! 1-95504,137265,137274,140059,140091,140117
alt.test! 1-1441498
```

Each line sets properties for the newsgroup named in the first field. The name is immediately followed by a character that indicates whether the owning user is currently subscribed to the group or not; a colon indicates subscription, and an exclamation mark indicates nonsubscription. The

[52] Password files are normally read once per user session at login time, and after that occasionally by file-system utilities like ls(1) that must map from numeric user and group IDs to names.

remainder of the line is a sequence of comma-separated article numbers or ranges of article numbers, indicating which articles the user has seen.

Non-Unix programmers might have automatically tried to design a fast binary format in which each newsgroup status was described by either a long but fixed-length binary record, or a sequence of self-describing binary packets with internal length fields. The main point of such a binary representation would be to express ranges with binary data in paired word-length fields, in order to avoid the overhead of parsing all the range expressions at startup.

Such a layout could be read and written faster than a textual format, but it would have other problems. A naïve implementation in fixed-length records would have placed artificial length limits on newsgroup names and (more seriously) on the maximum number of ranges of seen-article numbers. A more sophisticated binary-packet format would avoid the length limits, but could not be edited with the user's eyeballs and fingers — a capability that can be quite useful when you want to reset just some of the read bits in an individual newsgroup. Also, it would not necessarily be portable to different machine types.

The designers of the original newsreader chose transparency and interoperability over economy. The case for going in the other direction was not completely ridiculous; `.newsrc` files can get very large, and one modern reader (GNOME's Pan) uses a speed-optimized private format to avoid startup lag. But to other implementers, textual representation looked like a good tradeoff in 1980, and has looked better as machines increased in speed and storage dropped in price.

Case Study: The PNG Graphics File Format

PNG (Portable Network Graphics) is a file format for bitmap graphics. It is like GIF, and unlike JPEG, in that it uses lossless compression and is optimized for applications such as line art and icons rather than photographic images. Documentation and open-source reference libraries of high quality are available at the Portable Network Graphics website [http://www.libpng.org/pub/png/].

PNG is an excellent example of a thoughtfully designed binary format. A binary format is appropriate since graphics files may contain very large amounts of data, such that storage size and Internet download time would go up significantly if the pixel data were stored textually. Transaction economy was the prime consideration, with transparency sacrificed.[53] The designers were, however, careful about interoperability; PNG specifies byte orders, integer word lengths, endianness, and (lack of) padding between fields.

[53]Confusingly, PNG supports a different kind of transparency — transparent pixels in the PNG image.

A PNG file consists of a sequence of chunks, each in a self-describing format beginning with the chunk type name and the chunk length. Because of this organization, PNG does not need a release number. New chunk types can be added at any time; the case of the first letter in the chunk type name informs PNG-using software whether or not each chunk can be safely ignored.

The PNG file header also repays study. It has been cleverly designed to make various common kinds of file corruption (e.g., by 7-bit transmission links, or mangling of CR and LF characters) easy to detect.

The PNG standard is precise, comprehensive, and well written. It could serve as a model for how to write file format standards.

Data File Metaformats

A data file metaformat is a set of syntactic and lexical conventions that is either formally standardized or sufficiently well established by practice that there are standard service libraries to handle marshaling and unmarshaling it.

Unix has evolved or adopted metaformats suitable for a wide range of applications. It is good practice to use one of these (rather than an idiosyncratic custom format) wherever possible. The benefits begin with the amount of custom parsing and generation code that you may be able to avoid writing by using a service library. But the most important benefit is that developers and even many users will instantly recognize these formats and feel comfortable with them, which reduces the friction costs of learning new programs.

In the following discussion, when we refer to “traditional Unix tools” we are intending the combination of grep(1), sed(1), awk(1), tr(1), and cut(1) for doing text searches and transformations. Perl and other scripting languages tend to have good native support for parsing the line-oriented formats that these tools encourage.

Here, then, are the standard formats that can serve you as models.

DSV Style

DSV stands for *Delimiter-Separated Values*. Our first case study in textual metaformats was the `/etc/passwd` file, which is a DSV format with colon as the value separator. Under Unix, colon is the default separator for DSV formats in which the field values may contain whitespace.

`/etc/passwd` format (one record per line, colon-separated fields) is very traditional under Unix and frequently used for tabular data. Other classic examples include the `/etc/group` file describing security groups and the `/etc/inittab` file used to control startup and shutdown of Unix service programs at different run levels of the operating system.

Data files in this style are expected to support inclusion of colons in the data fields by backslash escaping. More generally, code that reads them is expected to support record continuation by ignoring backslash-escaped newlines, and to allow embedding nonprintable character data by C-style backslash escapes.

This format is most appropriate when the data is tabular, keyed by a name (in the first field), and records are typically short (less than 80 characters long). It works well with traditional Unix tools.

One occasionally sees field separators other than the colon, such as the pipe character | or even an ASCII NUL. Old-school Unix practice used to favor tabs, a preference reflected in the defaults for cut(1) and paste(1); but this has gradually changed as format designers became aware of the many small irritations that ensue from the fact that tabs and spaces are not visually distinguishable.

This format is to Unix what CSV (comma-separated value) format is under Microsoft Windows and elsewhere outside the Unix world. CSV (fields separated by commas, double quotes used to escape commas, no continuation lines) is rarely found under Unix.

In fact, the Microsoft version of CSV is a textbook example of how *not* to design a textual file format. Its problems begin with the case in which the separator character (in this case, a comma) is found inside a field. The Unix way would be to simply escape the separator with a backslash, and have a double escape represent a literal backslash. This design gives us a single special case (the escape character) to check for when parsing the file, and only a single action when the escape is found (treat the following character as a literal). The latter conveniently not only handles the separator character, but gives us a way to handle the escape character and newlines for free. CSV, on the other hand, encloses the entire field in double quotes if it contains the separator. If the field contains double quotes, it must also be enclosed in double quotes, and the individual double quotes in the field must themselves be repeated twice to indicate that they don't end the field.

The bad results of proliferating special cases are twofold. First, the complexity of the parser (and its vulnerability to bugs) is increased. Second, because the format rules are complex and underspecified, different implementations diverge in their handling of edge cases. Sometimes continuation lines *are* supported, by starting the last field of the line with an unterminated double quote — but only in some products! Microsoft has incompatible versions of CSV files between its

own applications, and in some cases between different versions of the same application (Excel being the obvious example here).

RFC 822 Format

The RFC 822 metaformat derives from the textual format of Internet electronic mail messages; RFC 822 is the principal Internet RFC describing this format (since superseded by RFC 2822). MIME (Multipurpose Internet Media Extension) provides a way to embed typed binary data within RFC-822-format messages. (Web searches on either of these names will turn up the relevant standards.)

In this metaformat, record attributes are stored one per line, named by tokens resembling mail header-field names and terminated with a colon followed by whitespace. Field names do not contain whitespace; conventionally a dash is substituted instead. The attribute value is the entire remainder of the line, exclusive of trailing whitespace and newline. A physical line that begins with tab or whitespace is interpreted as a continuation of the current logical line. A blank line may be interpreted either as a record terminator or as an indication that unstructured text follows.

Under Unix, this is the traditional and preferred textual metaformat for attributed messages or anything that can be closely analogized to electronic mail. More generally, it's appropriate for records with a varying set of fields in which the hierarchy of data is flat (no recursion or tree structure).

Usenet news uses it; so do the HTTP 1.1 (and later) formats used by the World Wide Web. It is very convenient for editing by humans. Traditional Unix search tools are still good for attribute searches, though finding record boundaries will be a little more work than in a record-per-line format.

One weakness of RFC 822 format is that when more than one RFC 822 message or record is put in a file, the record boundaries may not be obvious — how is a poor literal-minded computer to know where the unstructured text body of a message ends and the next header begins? Historically, there have been several different conventions for delimiting messages in mailboxes. The oldest and most widely supported, leading each message with a line that begins with the string `"From "` and sender information, is not appropriate for other kinds of records; it also requires that lines in message text beginning with `"From "` be escaped (typically with `>`) — a practice which not infrequently leads to confusion.

Some mail systems use delimiter lines consisting of control characters unlikely to appear in messages, such as several ASCII 01 (control-A) characters in succession. The MIME standard gets around the problem by including an explicit message length in the header, but this is a fragile

solution which is very likely to break if messages are ever manually edited. For a somewhat better solution, see the record-jar style described later in this chapter.

For examples of RFC 822 format, look in your mailbox.

Cookie-Jar Format

Cookie-jar format is used by the fortune(1) program for its database of random quotes. It is appropriate for records that are just bags of unstructured text. It simply uses newline followed by `%%` (or sometimes newline followed by `%`) as a record separator. Example 5.3 is an example section from a file of email signature quotes:

Example 5.3. A fortune file example.

```
"Among the many misdeeds of British rule in India, history will look
upon the Act depriving a whole nation of arms as the blackest."
      -- Mohandas Gandhi, "An Autobiography", pg 446
%
The people of the various provinces are strictly forbidden to have
in their possession any swords, short swords, bows, spears, firearms,
or other types of arms. The possession of unnecessary implements
makes difficult the collection of taxes and dues and tends to foment
uprisings.
      -- Toyotomi Hideyoshi, dictator of Japan, August 1588
%
"One of the ordinary modes, by which tyrants accomplish their
purposes without resistance, is, by disarming the people, and making
it an offense to keep arms."
      -- Supreme Court Justice Joseph Story, 1840
```

It is good practice to accept whitespace after `%` when looking for record delimiters. This helps cope with human editing mistakes. It’s even better practice to use `%%`, and ignore all text from `%%` to end-of-line.

> The cookie-jar separator was originally `%%\n`. I wanted something a bit more visible than `%` would have been. In fact, any stuff after the `%%` is treated as a comment (or at least that’s how I wrote it).
>
> —

<author>KenArnold</author>

Simple cookie-jar format is appropriate for pieces of text that have no natural ordering, distinguishable structure above word level, or search keys other than their text context.

Record-Jar Format

Cookie-jar record separators combine well with the RFC 822 metaformat for records, yielding a format we'll call 'record-jar'. If you need a textual format that will support multiple records with a variable repertoire of explicit fieldnames, one of the least surprising and human-friendliest ways to do it would look like Example 5.4.

Example 5.4. Basic data for three planets in a record-jar format.

```
Planet: Mercury
Orbital-Radius: 57,910,000 km
Diameter: 4,880 km
Mass: 3.30e23 kg
%%
Planet: Venus
Orbital-Radius: 108,200,000 km
Diameter: 12,103.6 km
Mass: 4.869e24 kg
%%
Planet: Earth
Orbital-Radius: 149,600,000 km
Diameter: 12,756.3 km
Mass: 5.972e24 kg
Moons: Luna
```

Of course, the record delimiter could be a blank line, but a line consisting of "`%%\n`" is more explicit and less likely to be introduced by accident during editing (two printable characters are better than one because it can't be generated by a single-character typo). In a format like this it is good practice to simply ignore blank lines.

If your records have an unstructured text part, your record-jar format is closely approaching a mailbox format. In this case, it's important that you have a well-defined way to escape the record delimiter so it can appear in text; otherwise, your record reader is going to choke on an ill-formed

text part someday. Some technique analogous to byte-stuffing (described later in this chapter) is indicated.

Record-jar format is appropriate for sets of field-attribute associations that are like DSV files, but have a variable repertoire of fields, and possibly unstructured text associated with them.

XML

XML is a very simple syntax resembling HTML — angle-bracketed tags and ampersand-led literal sequences. It is about as simple as a plain-text markup can be and yet express recursively nested data structures. XML is just a low-level syntax; it requires a document type definition (such as XHTML) and associated application logic to give it semantics.

XML is well suited for complex data formats (the sort of things for which the old-school Unix tradition would use an RFC-822-like stanza format) though overkill for simpler ones. It is especially appropriate for formats that have a complex nested or recursive structure of the sort that the RFC 822 metaformat does not handle well. For a good introduction to the format, see *XML in a Nutshell* [Harold-Means].

> Among the hardest things to get right in designing any text file format are issues of quoting, whitespace and other low-level syntax details. Custom file formats often suffer from slightly broken syntax that doesn't quite match other similar formats. Using a standard format such as XML, which is verifiable and parsed by a standard library, eliminates most of these issues.
>
> —
>
> <author>KeithPackard</author>

Example 5.5 is a simple example of an XML-based configuration file. It is part of the *kdeprint* tool shipped with the open-source KDE office suite hosted under Linux. It describes options for an image-to-PostScript filtering operation, and how to map them into arguments for a filter command. For another instructive example, see the discussion of *Glade* in Chapter 8.

Example 5.5. An XML example.

```
<?xml version="1.0"?>
<kprintfilter name="imagetops">
   <filtercommand
```

```
        data="imagetops %filterargs %filterinput %filteroutput" />
   <filterargs>
      <filterarg name="center"
              description="Image centering"
              format="-nocenter" type="bool" default="true">
         <value name="true" description="Yes" />
         <value name="false" description="No" />
      </filterarg>
      <filterarg name="turn"
              description="Image rotation"
              format="-%value" type="list" default="auto">
         <value name="auto" description="Automatic" />
         <value name="noturn" description="None" />
         <value name="turn" description="90 deg" />
      </filterarg>
      <filterarg name="scale"
              description="Image scale"
              format="-scale %value"
              type="float"
                 min="0.0" max="1.0" default="1.000" />
      <filterarg name="dpi"
              description="Image resolution"
              format="-dpi %value"
              type="int" min="72" max="1200" default="300" />
   </filterargs>
   <filterinput>
      <filterarg name="file" format="%in" />
      <filterarg name="pipe" format="" />
   </filterinput>
   <filteroutput>
      <filterarg name="file" format="> %out" />
      <filterarg name="pipe" format="" />
   </filteroutput>
</kprintfilter>
```

One advantage of XML is that it is often possible to detect ill-formed, corrupted, or incorrectly generated data through a syntax check, without knowing the semantics of the data.

The most serious problem with XML is that it doesn't play well with traditional Unix tools. Software that wants to read an XML format needs an XML parser; this means bulky, complicated programs. Also, XML is itself rather bulky; it can be difficult to see the data amidst all the markup.

One application area in which XML is clearly winning is in markup formats for document files (we'll have more to say about this in Chapter 18). Tagging in such documents tends to be relatively sparse among large blocks of plain text; thus, traditional Unix tools still work fairly well for simple text searches and transformations.

One interesting bridge between these worlds is PYX format — a line-oriented translation of XML that can be hacked with traditional line-oriented Unix text tools and then losslessly translated back to XML. A Web search for "Pyxie" will turn up resources. The xmltk toolkit takes the opposite tack, providing stream-oriented tools analogous to grep(1) and sort(1) for filtering XML documents; Web search for "xmltk" to find it.

XML can be a simplifying choice or a complicating one. There is a lot of hype surrounding it, but don't become a fashion victim by either adopting or rejecting it uncritically. Choose carefully and bear the KISS principle in mind.

Windows INI Format

Many Microsoft Windows programs use a textual data format that looks like Example 5.6. This example associates optional resources named `account`, `directory`, `numeric_id`, and `developer` with named projects `python`, `sng`, `fetchmail`, and `py-howto`. The DEFAULT entry supplies values that will be used when a named entry fails to supply them.

Example 5.6. A `.INI` file example.

```
[DEFAULT]
account = esr

[python]
directory = /home/esr/cvs/python/
developer = 1

[sng]
directory = /home/esr/WWW/sng/
numeric_id = 1012
```

```
developer = 1

[fetchmail]
numeric_id = 18364

[py-howto]
account = eric
directory = /home/esr/cvs/py-howto/
developer = 1
```

This style of data-file format is not native to Unix, but some Linux programs (notably Samba, the suite of tools for accessing Windows file shares from Linux) support it under Windows's influence. This format is readable and not badly designed, but like XML it doesn't play well with grep(1) or conventional Unix scripting tools.

The .INI format is appropriate if your data naturally falls into its two-level organization of name-attribute pairs clustered under named records or sections. It's not good for data with a fully recursive treelike structure (XML is more appropriate for that), and it would be overkill for a simple list of name-value associations (use DSV format for that).

Unix Textual File Format Conventions

There are long-standing Unix traditions about how textual data formats ought to look. Most of these derive from one or more of the standard Unix metaformats we've just described. It is wise to follow these conventions unless you have strong and specific reasons to do otherwise.

In Chapter 10 we will discuss a different set of conventions used for program run-control files, but you should notice that it will share some of these same rules (especially about the lexical level, the rules by which characters are assembled into tokens).

- *One record per newline-terminated line, if possible.* This makes it easy to extract records with text-stream tools. For data interchange with other operating systems, it's wise to make your file-format parser indifferent to whether the line ending is LF or CR-LF. It's also conventional to ignore trailing whitespace in such formats; this protects against common editor bobbles.

- *Less than 80 characters per line, if possible.* This makes the format browseable in an ordinary-sized terminal window. If many records must be longer than 80 characters, consider a stanza format (see below).

- *Use # as an introducer for comments.* It is good to have a way to embed annotations and comments in data files. It's best if they're actually part of the file structure, and so will be preserved by tools that know its format. For comments that are not preserved during parsing, # is the conventional start character.

- *Support the backslash convention.* The least surprising way to support embedding nonprintable control characters is by parsing C-like backslash escapes — `\n` for a newline, `\r` for a carriage return, `\t` for a tab, `\b` for backspace, `\f` for formfeed, `\e` for ASCII escape (27), `\nnn` or `\onnn` or `\0nnn` for the character with octal value `nnn`, `\xnn` for the character with hexadecimal value `nn`, `\dnnn` for the character with decimal value `nnn`, `\\` for a literal backslash. A newer convention, but one worth following, is the use of `\unnnn` for a hexadecimal Unicode literal.

- *In one-record-per-line formats, use colon or any run of whitespace as a field separator.* The colon convention seems to have originated with the Unix password file. If your fields must contain instances of the separator(s), use a backslash as the prefix to escape them.

- *Do not allow the distinction between tab and whitespace to be significant.* This is a recipe for serious headaches when the tab settings on your users' editors are different; more generally, it's confusing to the eye. Using tab alone as a field separator is especially likely to cause problems; allowing any run of tabs and spaces to be a field separator, on the other hand, works well.

- *Favor hex over octal.* Hex-digit pairs and quads are easier to eyeball-map into bytes and today's 32- and 64-bit words than octal digits of three bits each; also marginally more efficient. This rule needs emphasizing because some older Unix tools such as od(1) violate it; that's a legacy from the instruction field sizes in the machine languages of older PDP minicomputers.

- *For complex records, use a 'stanza' format: multiple lines per record, with a record separator line of `%%\n` or `%\n`.* The separators make useful visual boundaries for human beings eyeballing the file.

- *In stanza formats, either have one record field per line or use a record format resembling RFC 822 electronic-mail headers, with colon-terminated field-name keywords leading fields.* The second choice is appropriate when fields are often either absent or longer than 80 characters, or when records are sparse (e.g., often with empty fields).

- *In stanza formats, support line continuation.* When interpreting the file, either discard backslash followed by whitespace or interpret newline followed by whitespace equivalently to a single space, so that a long logical line can be folded into short (easily editable!) physical lines. It's also conventional to ignore trailing whitespace in these formats; this convention protects against common editor bobbles.

- *Either include a version number or design the format as self-describing chunks independent of each other.* If there is even the faintest possibility that the format will have to be changed or extended, include a version number so your code can conditionally do the right thing on all versions. Alternatively, design the format as self-describing chunks so that you can add new chunk types without instantly breaking old code.

- *Beware of floating-point round-off problems.* Conversion of floating-point numbers from binary to text format and back can lose precision, depending on the quality of the conversion library you are using. If the structure you are marshaling/unmarshaling contains floating point, you should test the conversion in both directions. If it looks like conversion in either direction is subject to roundoff errors, be prepared to dump the floating-point field as raw binary instead, or a string encoding thereof. If you're coding in C or some language that has access to C printf/scanf, the C99 `%a` specifier may solve this problem.

- *Don't bother compressing or binary-encoding just part of the file.* See below...

The Pros and Cons of File Compression

Many modern Unix projects, such as OpenOffice.org and AbiWord, now use XML compressed with zip(1) or gzip(1) as a data file format. Compressed XML combines space economy with some of the advantages of a textual format — notably, it avoids the problem that binary formats must often allocate space for information that may not be used in particular cases (e.g., for unusual options or large ranges). But there is some dispute about this, dispute which turns on some of the central tradeoffs discussed in this chapter.

On the one hand, experiments have shown that documents in a compressed XML file are usually significantly smaller than the Microsoft Word's native file format, a binary format that one might imagine would take less space. The reason relates to a fundamental of the Unix philosophy: Do one thing well. Creating a single tool to do the compression job well is more effective than ad-hoc compression on parts of the file, because the tool can look across all the data and exploit *all* repetition in the information.

Also, by separating the representation design from the particular compression method used, you leave open the possibility of using different compression methods in the future with no more than minimal changes to the actual file parsing — perhaps, with no changes at all.

On the other hand, compression does some damage to transparency. While a human being can estimate from context whether uncompressing the file is likely to show him anything useful, tools such as file(1) cannot as of mid-2003 see through the wrapping.

Some would advocate a less structured compression format — straight gzip(1)-compressed XML data, say, without the internal structure and self-identifying header chunk provided by zip(1). While using a format similar to that of zip(1) solves the identification problem, it means that decoding such files will be tricky for programs written in the simpler scripting languages.

Any of these solutions (straight text, straight binary, or compressed text) may be optimal depending on the relative weight you give to storage economy, discoverability, or making browsing tools as simple as possible to write. The point of the preceding discussion is not to advocate any one of these approaches over the others, but rather to suggest how you can think about the options and design tradeoffs clearly.

This having been said, the truly Unixy solution would probably be to fix file(1) to see file prefixes through the compression — and, failing that, to write a shellscript wrapper around file(1) that would interpret compression as a direction to apply gunzip(1) and take a second look.

Application Protocol Design

In Chapter 7, we'll discuss the advantages of breaking complicated applications up into cooperating processes speaking an application-specific command set or protocol with each other. All the good reasons for data file formats to be textual apply to these application-specific protocols as well.

When your application protocol is textual and easily parsed by eyeball, many good things become easier. Transaction dumps become much easier to interpret. Test loads become easier to write.

Server processes are often invoked by harness programs such as inetd(8) in such a way that the server sees commands on standard input and ships responses to standard output. We describe this "CLI server" pattern in more detail in Chapter 11.

A CLI server with a command set that is designed for simplicity has the valuable property that a human tester will be able to type commands direct to the server process to probe the software's behavior.

Another issue to bear in mind is the end-to-end design principle. Every protocol designer should read the classic *End-to-End Arguments in System Design* [Saltzer]. There are often serious questions about which level of the protocol stack should handle features like security and authentication; this paper provides some good conceptual tools for thinking about them. Yet a third issue is designing application protocols for good performance. We'll cover that issue in more detail in Chapter 12.

The traditions of Internet application protocol design evolved separately from Unix before 1980.[54] But since the 1980s these traditions have become thoroughly naturalized into Unix practice.

We'll illustrate the Internet style by looking at three application protocols that are both among the most heavily used, and are widely regarded among Internet hackers as paradigmatic: SMTP, POP3, and IMAP. All three address different aspects of mail transport (one of the net's two most important applications, along with the World Wide Web), but the problems they address (passing messages, setting remote state, indicating error conditions) are generic to non-email application protocols as well and are normally addressed using similar techniques.

Case Study: SMTP, a Simple Socket Protocol

Example 5.7 is an example transaction in SMTP (Simple Mail Transfer Protocol), which is described by RFC 2821. In the example, *C:* lines are sent by a mail transport agent (MTA) sending mail, and *S:* lines are returned by the MTA receiving it. Text *emphasized like this* is comments, not part of the actual transaction.

Example 5.7. An SMTP session example.

```
C: <client connects to service port 25>
C: HELO snark.thyrsus.com           sending host identifies self
S: 250 OK Hello snark, glad to meet you  receiver acknowledges
C: MAIL FROM: <esr@thyrsus.com>       identify sending user
S: 250 <esr@thyrsus.com>... Sender ok   receiver acknowledges
C: RCPT TO: cor@cpmy.com            identify target user
```

[54] One relic of this pre-Unix history is that Internet protocols normally use CR-LF as a line terminator rather than Unix's bare LF.

```
S: 250 root... Recipient ok          receiver acknowledges
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Scratch called.  He wants to share
C: a room with us at Balticon.
C: .                               end of multiline send
S: 250 WAA01865 Message accepted for delivery
C: QUIT                             sender signs off
S: 221 cpmy.com closing connection     receiver disconnects
C: <client hangs up>
```

This is how mail is passed among Internet machines. Note the following features: command-argument format of the requests, responses consisting of a status code followed by an informational message, the fact that the payload of the DATA command is terminated by a line consisting of a single dot.

SMTP is one of the two or three oldest application protocols still in use on the Internet. It is simple, effective, and has withstood the test of time. The traits we have called out here are tropes that recur frequently in other Internet protocols. If there is any single archetype of what a well-designed Internet application protocol looks like, SMTP is it.

Case Study: POP3, the Post Office Protocol

Another one of the classic Internet protocols is POP3, the Post Office Protocol. It is also used for mail transport, but where SMTP is a ‘push’ protocol with transactions initiated by the mail sender, POP3 is a ‘pull’ protocol with transactions initiated by the mail receiver. Internet users with intermittent access (like dial-up connections) can let their mail pile up on a mail-drop machine, then use a POP3 connection to pull mail up the wire to their personal machines.

Example 5.8 is an example POP3 session. In the example, *C:* lines are sent by the client, and *S:* lines by the mail server. Observe the many similarities with SMTP. This protocol is also textual and line-oriented, sends payload message sections terminated by a line consisting of a single dot followed by line terminator, and even uses the same exit command, QUIT. Like SMTP, each client operation is acknowledged by a reply line that begins with a status code and includes an informational message meant for human eyes.

Example 5.8. A POP3 example session.

```
C: <client connects to service port 110>
S: +OK POP3 server ready <1896.6971@mailgate.dobbs.org>
C: USER bob
S: +OK bob
C: PASS redqueen
S: +OK bob's maildrop has 2 messages (320 octets)
C: STAT
S: +OK 2 320
C: LIST
S: +OK 2 messages (320 octets)
S: 1 120
S: 2 200
S: .
C: RETR 1
S: +OK 120 octets
S: <the POP3 server sends the text of message 1>
S: .
C: DELE 1
S: +OK message 1 deleted
C: RETR 2
S: +OK 200 octets
S: <the POP3 server sends the text of message 2>
S: .
C: DELE 2
S: +OK message 2 deleted
C: QUIT
S: +OK dewey POP3 server signing off (maildrop empty)
C: <client hangs up>
```

There are a few differences. The most obvious one is that POP3 uses status tokens rather than SMTP's 3-digit status codes. Of course the requests have different semantics. But the family resemblance (one we'll have more to say about when we discuss the generic Internet metaprotocol later in this chapter) is clear.

Case Study: IMAP, the Internet Message Access Protocol

To complete our triptych of Internet application protocol examples, we'll look at IMAP, another post office protocol designed in a slightly different style. See Example 5.9; as before, *C:* lines are sent by the client, and *S:* lines by the mail server. Text *emphasized like this* is comments, not part of the actual transaction.

Example 5.9. An IMAP session example.

```
C: <client connects to service port 143>
S: * OK example.com IMAP4rev1 v12.264 server ready
C: A0001 USER "frobozz" "xyzzy"
S: * OK User frobozz authenticated
C: A0002 SELECT INBOX
S: * 1 EXISTS
S: * 1 RECENT
S: * FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
S: * OK [UNSEEN 1] first unseen message in /var/spool/mail/esr
S: A0002 OK [READ-WRITE] SELECT completed
C: A0003 FETCH 1 RFC822.SIZE               Get message sizes
S: * 1 FETCH (RFC822.SIZE 2545)
S: A0003 OK FETCH completed
C: A0004 FETCH 1 BODY[HEADER]              Get first message header
S: * 1 FETCH (RFC822.HEADER {1425}
<server sends 1425 octets of message payload>
S: )
S: A0004 OK FETCH completed
C: A0005 FETCH 1 BODY[TEXT]                Get first message body
S: * 1 FETCH (BODY[TEXT] {1120}
<server sends 1120 octets of message payload>
S: )
S: * 1 FETCH (FLAGS (\Recent \Seen))
S: A0005 OK FETCH completed
C: A0006 LOGOUT
S: * BYE example.com IMAP4rev1 server terminating connection
S: A0006 OK LOGOUT completed
C: <client hangs up>
```

IMAP delimits payloads in a slightly different way. Instead of ending the payload with a dot, the payload length is sent just before it. This increases the burden on the server a little bit (messages

have to be composed ahead of time, they can't just be streamed up after the send initiation) but makes life easier for the client, which can tell in advance how much storage it will need to allocate to buffer the message for processing as a whole.

Also, notice that each response is tagged with a sequence label supplied by the request; in this example they have the form A000n, but the client could have generated any token into that slot. This feature makes it possible for IMAP commands to be streamed to the server without waiting for the responses; a state machine in the client can then simply interpret the responses and payloads as they come back. This technique cuts down on latency.

IMAP (which was designed to replace POP3) is an excellent example of a mature and powerful Internet application protocol design, one well worth study and emulation.

Application Protocol Metaformats

Just as data file metaformats have evolved to simplify serialization for storage, application protocol metaformats have evolved to simplify serialization for transactions across networks. The tradeoffs are a little different in this case; because network bandwidth is more expensive than storage, there is more of a premium on transaction economy. Still, the transparency and interoperability benefits of textual formats are sufficiently strong that most designers have resisted the temptation to optimize for performance at the cost of readability.

The Classical Internet Application Metaprotocol

Marshall Rose's RFC 3117, *On the Design of Application Protocols*,[55] provides an excellent overview of the design issues in Internet application protocols. It makes explicit several of the tropes in classical Internet application protocols that we observed in our examination of SMTP, POP, and IMAP, and provides an instructive taxonomy of such protocols. It is recommended reading.

The classical Internet metaprotocol is textual. It uses single-line requests and responses, except for payloads which may be multiline. Payloads are shipped either with a preceding length in octets or with a terminator that is the line `".\r\n"`. In the latter case the payload is *byte-stuffed*; all lines that start with a period get another period prepended, and the receiver side is responsible for both recognizing the termination and stripping away the stuffing. Response lines consist of a status code followed by a human-readable message.

[55] See RFC 3117 [ftp://ftp.rfc-editor.org/in-notes/rfc3117.txt].

One final advantage of this classical style is that it is readily extensible. The parsing and state-machine framework doesn't need to change much to accommodate new requests, and it is easy to code implementations so that they can parse unknown requests and return an error or simply ignore them. SMTP, POP3, and IMAP have all been extended in minor ways fairly often during their lifetimes, with minimal interoperability problems. Naïvely designed binary protocols are, by contrast, notoriously brittle.

HTTP as a Universal Application Protocol

Ever since the World Wide Web reached critical mass around 1993, application protocol designers have shown an increasing tendency to layer their special-purpose protocols on top of HTTP, using web servers as generic service platforms.

This is a viable option because, at the transaction layer, HTTP is very simple and general. An HTTP request is a message in an RFC-822/MIME-like format; typically, the headers contain identification and authentication information, and the first line is a method call on some resource specified by a Universal Resource Indicator (URI). The most important methods are GET (fetch the resource), PUT (modify the resource) and POST (ship data to a form or back-end process). The most important form of URI is a URL or Uniform Resource Locator, which identifies the resource by service type, host name, and a location on the host. An HTTP response is simply an RFC-822/MIME message and can contain arbitrary content to be interpreted by the client.

Web servers handle the transport and request-multiplexing layers of HTTP, as well as standard service types like http and ftp. It is relatively easy to write web server plugins that will handle custom service types, and to dispatch on other elements of the URI format.

Besides avoiding a lot of lower-level details, this method means the application protocol will tunnel through the standard HTTP service port and not need a TCP/IP service port of its own. This can be a distinct advantage; most firewalls leave port 80 open, but trying to punch another hole through can be fraught with both technical and political difficulties.

With this advantage comes a risk. It means that your web server and its plugins grow more complex, and cracks in any of that code can have large security implications. It may become more difficult to isolate and shut down problem services. The usual tradeoffs between security and convenience apply.

RFC 3205, *On the Use of HTTP As a Substrate,*[56] has good design advice for anyone considering using HTTP as the underlayer of an application protocol, including a summary of the tradeoffs and problems involved.

Case Study: The `CDDB/freedb.org` Database

Audio CDs consist of a sequence of music tracks in a digital format called CDDA-WAV. They were designed to be played by very simple consumer-electronics devices a few years before general-purpose computers developed enough raw speed and sound capability to decode them on the fly. Because of this, there is no provision in the format for even simple metainformation such as the album and track titles. But modern computer-hosted CD players want this information so the user can assemble and edit play lists.

Enter the Internet. There are (at least two) repositories that provide a mapping between a hash code computed from the track-length table on a CD and artist/album-title/track-title records. The original was `cddb.org`, but another site called `freedb.org` which is probably now more complete and widely used. Both sites rely on their users for the enormous task of keeping the database current as new CDs come out; `freedb.org` arose from a developer revolt after CDDB elected to take all that user-contributed information proprietary .

Queries to these services could have been implemented as a custom application protocol on top of TCP/IP, but that would have required steps such as getting a new TCP/IP port number assigned and fighting to get a hole for it punched through thousands of firewalls. Instead, the service is implemented over HTTP as a simple CGI query (as if the CD's hash code had been supplied by a user filling in a Web form).

This choice makes all the existing infrastructure of HTTP and Web-access libraries in various programming languages available to support programs for querying and updating this database. As a result, adding such support to a software CD player is nearly trivial, and effectively every software CD player knows how to use them.

Case Study: Internet Printing Protocol

Internet Printing Protocol (IPP) is a successful, widely implemented standard for the control of network-accessible printers. Pointers to RFCs, implementations, and much other related material are available at the IETF's Printer Working Group [http://www.pwg.org/ipp/] site.

[56] See RFC 3205 [http://www.faqs.org/rfcs/rfc3205.html].

IPP uses HTTP 1.1 as a transport layer. All IPP requests are passed via an HTTP POST method call; responses are ordinary HTTP responses. (Section 4.2 of RFC 2568, *Rationale for the Structure of the Model and Protocol for the Internet Printing Protocol*, does an excellent job of explaining this choice; it repays study by anyone considering writing a new application protocol.)

From the software side, HTTP 1.1 is widely deployed. It already solves many of the transport-level problems that would otherwise distract protocol developers and implementers from concentrating on the domain semantics of printing. It is cleanly extensible, so there is room for IPP to grow. The CGI programming model for handling the POST requests is well understood and development tools are widely available.

Most network-aware printers already embed a web server, because that's the natural way to make the status of the printer remotely queryable by human beings. Thus, the incremental cost of adding IPP service to the printer firmware is not large. (This is an argument that could be applied to a remarkably wide range of other network-aware hardware, including vending machines and coffee makers[57] and hot tubs!)

About the only serious drawback of layering IPP over HTTP is that the protocol is completely driven by client requests. Thus there is no space in the model for printers to ship asynchronous alert messages back to clients. (However, smarter clients could run a trivial HTTP server to receive such alerts formatted as HTTP requests from the printer.)

BEEP: Blocks Extensible Exchange Protocol

BEEP (formerly BXXP) is a generic protocol machine that competes with HTTP for the role of universal underlayer for application protocols. There is a niche open because there is not as yet any other more established metaprotocol that is appropriate for truly peer-to-peer applications, as opposed to the client-server applications that HTTP handles well. A project website [http://www.beepcore.org/beepcore/docs/sl-beep.jsp] provides access to standards and open-source implementations in several languages.

BEEP has features to support both client-server and peer-to-peer modes. The authors designed the BEEP protocol and support library so that picking the right options abstracts away messy issues like data encoding, flow control, congestion-handling, support of end-to-end encryption, and assembling a large response composed of multiple transmissions,

[57] See RFC 2324 [http://www.ietf.org/rfc/rfc2324.txt] and RFC 2325 [http://www.ietf.org/rfc/rfc2325.txt].

Internally, BEEP peers exchange sequences of self-describing binary packets not unlike chunk types in PNG. The design is tuned more for economy and less for transparency than the classical Internet protocols or HTTP, and might be a better choice when data volumes are large. BEEP also avoids the HTTP problem that all requests have to be client-initiated; it would be better in situations in which a server needs to send asynchronous status messages back to the client.

BEEP is still new technology in mid-2003, and has only a few demonstration projects. But the BEEP papers are good analytical surveys of best practice in protocol design; even if BEEP itself fails to gain widespread adoption, the papers will retain considerable tutorial value.

XML-RPC, SOAP, and Jabber

There is a developing trend in application protocol design toward using XML within MIME to structure requests and payloads. BEEP peers use this format for channel negotiations. Three major protocols are going the XML route throughout: XML-RPC and SOAP (Simple Object Access Protocol) for remote procedure calls, and Jabber for instant messaging and presence. All three are XML document types.

XML-RPC is very much in the Unix spirit (its author observes that he learned how to program in the 1970s by reading the original source code for Unix). It's deliberately minimalist but nevertheless quite powerful, offering a way for the vast majority of RPC applications that can get by on passing around scalar boolean/integer/float/string datatypes to do their thing in a way that is lightweight and easy to understand and monitor. XML-RPC's type ontology is richer than that of a text stream, but still simple and portable enough to act as a valuable check on interface complexity. Open-source implementations are available. An excellent XML-RPC home page [http://www.xmlrpc.com/] points to specifications and multiple open-source implementations.

SOAP is a more heavyweight RPC protocol with a richer type ontology that includes arrays and C-like structs. It was inspired by XML-RPC, but has been plausibly accused of being an overdesigned victim of the second-system effect. As of mid-2003 the SOAP standard is still a work in progress, but a trial implementation in Apache is tracking the drafts. Open-source client modules in Perl, Python, Tcl, and Java are readily discoverable by a Web search. The W3C draft specification is available on the Web [http://www.w3.org/TR/SOAP/].

XML-RPC and SOAP, considered as remote procedure call methods, have some associated risks that we discuss at the end of Chapter 7.

Jabber is a peer-to-peer protocol designed to support instant messaging and presence. What makes it interesting as an application protocol is that it supports passing around XML forms and live documents. Specifications, documentation, and open-source implementations are available at the Jabber Software Foundation [http://www.jabber.org/about/overview.html] site.

Chapter 6. Transparency

Let There Be Light

Beauty is more important in computing than anywhere else in technology because software is so complicated. Beauty is the ultimate defense against complexity.
--
<author>DavidGelernter</author>
Machine Beauty: Elegance and the Heart of Technology (1998)

In the previous chapter we discussed the importance of textual data formats and application protocols, representations that are easy for human beings to examine and interact with. These promote qualities in design that are much valued in the Unix tradition but seldom if ever talked about explicitly: *transparency* and *discoverability*.

Software systems are transparent when they don't have murky corners or hidden depths. Transparency is a passive quality. A program is transparent when it is possible to form a simple mental model of its behavior that is actually predictive for all or most cases, because you can see through the machinery to what is actually going on.

Software systems are discoverable when they include features that are designed to help you build in your mind a correct mental model of what they do and how they work. So, for example, good documentation helps discoverability to a user. Good choice of variable and function names helps discoverability to a programmer. Discoverability is an active quality. To achieve it in your software you cannot merely fail to be obscure, you have to go out of your way to be helpful.[58]

Transparency and discoverability are important for both users and software developers. But they're important in different ways. Users like these properties in a UI because they mean an easier learning curve. UI transparency and discoverability are a large part of what people mean when they say a UI is 'intuitive'; most of the rest is the Rule of Least Surprise. We'll examine the properties that make user interfaces pleasant and effective in more depth in Chapter 11.

Software developers like these qualities in the code itself (the part users don't see) because they so often need to understand it well enough to modify and debug it. Also, a program designed so that its internal data flows are readily comprehensible is more likely to be one that does not fail because

[58] An economically-minded friend comments: "Discoverability is about reducing barriers to entry; transparency is about reducing the cost of living in the code".

of bad interactions that the designer didn't notice, and more likely to be able to evolve forward gracefully (including accommodating change when new maintainers pick up the baton).

Transparency is a major component of what David Gelernter refers to as "beauty" in this chapter's epigraph. Unix programmers, borrowing from mathematicians, often use the more specific term "elegance" for the quality Gelernter speaks of. Elegance is a combination of power and simplicity. Elegant code does much with little. Elegant code is not only correct but visibly, *transparently* correct. It does not merely communicate an algorithm to a computer, but also conveys insight and assurance to the mind of a human that reads it. By seeking elegance in our code, we build better code. Learning to write transparent code is a first, long step toward learning how to write elegant code — and taking care to make code discoverable helps us learn how to make it transparent. Elegant code is both transparent and discoverable.

It may be easier to appreciate the difference between transparency and discoverability with a pair of extreme examples. The Linux kernel source is remarkably transparent (given the intrinsic complexity of what it does) but not at all discoverable — acquiring the minimum knowledge needed to live in the code and understand the idiom of the developers is difficult, but once you do the whole makes sense.[59] On the other hand, the Emacs Lisp libraries are discoverable but not transparent. It's easy to acquire enough knowledge to tweak just one thing, but quite difficult to comprehend the whole system.

In this chapter, we'll examine features of Unix designs that promote transparency and discoverability not just in UIs but in the parts users don't normally see. We'll develop some useful rules you can apply to your coding and development practice. Later on, in Chapter 19 we'll see how good release-engineering practices (like having a `README` file with appropriate content) can make your source code as discoverable as your design.

If you need a practical reminder why these qualities are important, remember that the sanity you save by writing transparent, discoverable systems may well be that of your own future self.

Studying Cases

Normal practice in this book has been to intersperse case studies with philosophy. But in this chapter we'll begin by looking at several Unix designs that exhibit transparency and discoverability,

[59]The Linux kernel makes a number of attempts at discoverability, including the Documentation subdirectory in the Linux kernel source tarball and quite a number of tutorial websites and books. These attempts are frustrated by the speed at which the kernel changes; the documentation has a chronic tendency to fall behind.

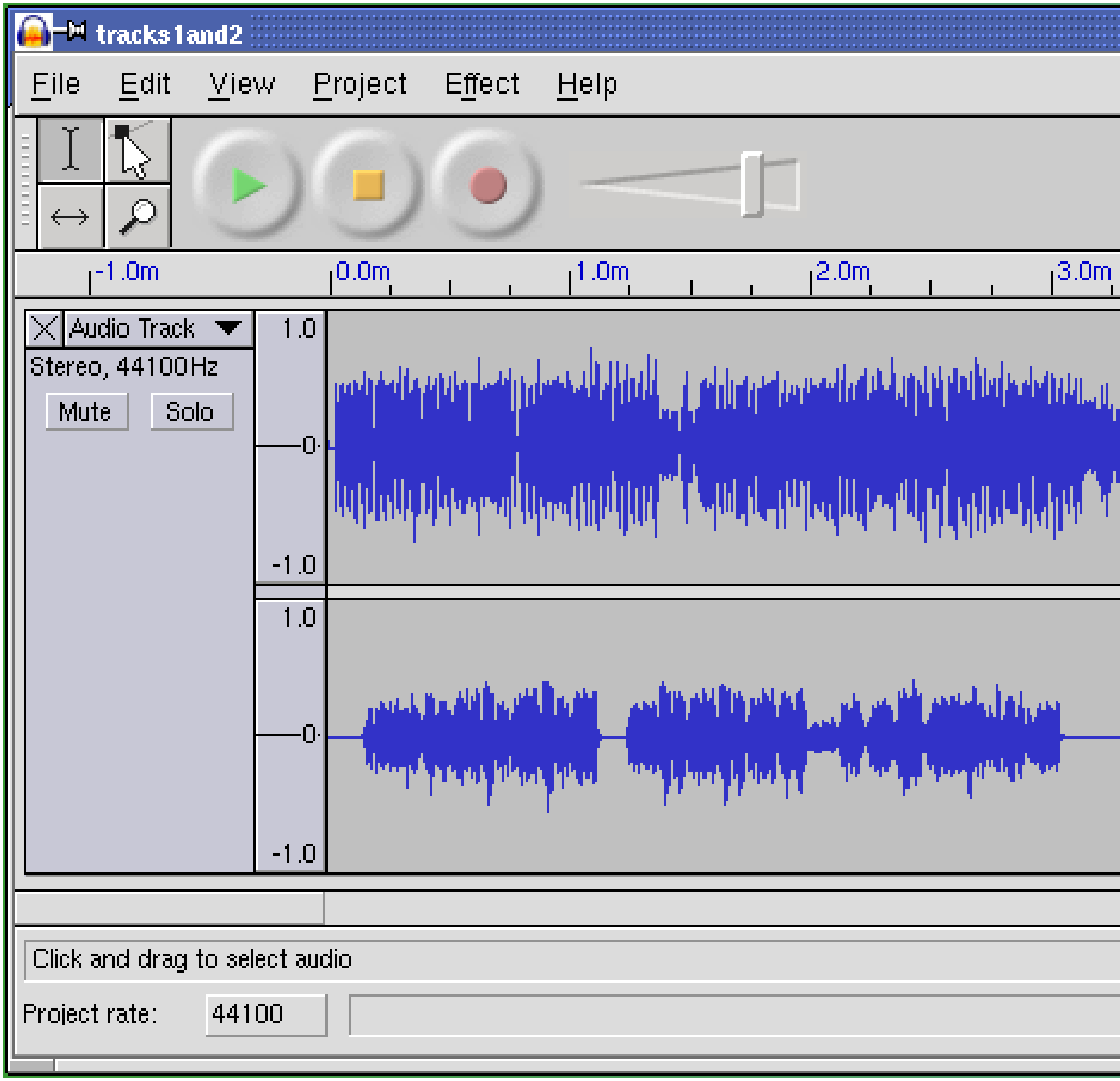
and attempt to draw lessons from them only after all have been presented. Each major point of the analysis in the latter half of this chapter draws on several of these, and the arrangement avoids forward references to case studies the reader hasn't seen yet.

Case Study: *audacity*

First, we'll look at an example of transparency in UI design. It is *audacity*, an open-source editor for sound files that runs on Unix systems, Mac OS X, and Windows. Sources, downloadable binaries, documentation, and screen shots are available at the project site [http://audacity.sourceforge.net/].

This program supports cutting, pasting, and editing of audio samples. It supports multitrack editing and mixing. The UI is superbly simple; the sound waveforms are shown in the *audacity* window. The image of the waveform can be cut and pasted; operations on that image are directly reflected in the audio sample as soon as they are performed.

Figure 6.1. Screen shot of *audacity*.



Multitrack editing is supported in the simplest possible way; the screen splits into multiple per-track displays in a spatial relationship that conveys their concurrency and makes it easy to match features by inspection. Tracks can be dragged right or left with the mouse to change their relative timing.

Several features of this UI are subtly excellent and worthy of emulation: the large, easily visible and clickable operation buttons with distinguishing colors, the presence of an undo command that removes most of the risk from experimentation, the volume slider that makes softness/loudness visually obvious in its shape.

But these are details. The central virtue of this program is that it has a superbly transparent and natural user interface, one that erects as few barriers between the user and the sound file as possible.

Case Study: *fetchmail*'s `-v` option

fetchmail is a network gateway program. Its main purpose is to translate between POP3 or IMAP remote-mail protocols and the Internet's native SMTP protocol for email exchange. It is in extremely widespread use on Unix machines that use intermittent SLIP or PPP connections to Internet service providers, and as such probably touches an appreciable fraction of the Internet's mail traffic.

fetchmail has no fewer than 60 command-line options (which, as we'll establish later in this book, is probably too many), and a number of other options that are settable from the run-control file but not from the command line. Of all these, the most important — by far — is `-v`, the verbose option.

When `-v` is on, *fetchmail* dumps each one of its POP, IMAP, and SMTP transactions to standard output as they happen. A developer can actually see the code doing protocol with remote mailservers and the mail transport program it forwards to, in real time. Users can send session transcripts with their bug reports. Example 6.1 shows a representative session transcript.

Example 6.1. An example *fetchmail* `-v` transcript.

```
fetchmail: 6.1.0 querying hurkle.thyrsus.com (protocol IMAP)
           at Mon, 09 Dec 2002 08:41:37 -0500 (EST): poll started
fetchmail: running ssh %h /usr/sbin/imapd
           (host hurkle.thyrsus.com service imap)
fetchmail: IMAP< * PREAUTH [42.42.1.0] IMAP4rev1 v12.264 server ready
fetchmail: IMAP> A0001 CAPABILITY
fetchmail: IMAP< * CAPABILITY IMAP4 IMAP4REV1 NAMESPACE IDLE SCAN
```

```
           SORT MAILBOX-REFERRALS LOGIN-REFERRALS AUTH=LOGIN
           THREAD=ORDEREDSUBJECT
fetchmail: IMAP< A0001 OK CAPABILITY completed
fetchmail: IMAP> A0002 SELECT "INBOX"
fetchmail: IMAP< * 2 EXISTS
fetchmail: IMAP< * 1 RECENT
fetchmail: IMAP< * OK [UIDVALIDITY 1039260713] UID validity status
fetchmail: IMAP< * OK [UIDNEXT 23982] Predicted next UID
fetchmail: IMAP< * FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
fetchmail: IMAP< * OK [PERMANENTFLAGS
           (\* \Answered \Flagged \Deleted \Draft \Seen)]
           Permanent flags
fetchmail: IMAP< * OK [UNSEEN 2] first unseen in /var/spool/mail/esr
fetchmail: IMAP< A0002 OK [READ-WRITE] SELECT completed
fetchmail: IMAP> A0003 EXPUNGE
fetchmail: IMAP< A0003 OK Mailbox checkpointed, no messages expunged
fetchmail: IMAP> A0004 SEARCH UNSEEN
fetchmail: IMAP< * SEARCH 2
fetchmail: IMAP< A0004 OK SEARCH completed
2 messages (1 seen) for esr at hurkle.thyrsus.com.
fetchmail: IMAP> A0005 FETCH 1:2 RFC822.SIZE
fetchmail: IMAP< * 1 FETCH (RFC822.SIZE 2545)
fetchmail: IMAP< * 2 FETCH (RFC822.SIZE 8328)
fetchmail: IMAP< A0005 OK FETCH completed
skipping message esr@hurkle.thyrsus.com:1 (2545 octets) not flushed
fetchmail: IMAP> A0006 FETCH 2 RFC822.HEADER
fetchmail: IMAP< * 2 FETCH (RFC822.HEADER {1586}
reading message esr@hurkle.thyrsus.com:2 of 2 (1586 header octets)
fetchmail: SMTP< 220 snark.thyrsus.com ESMTP Sendmail 8.12.5/8.12.5;
           Mon, 9 Dec
2002 08:41:41 -0500

fetchmail: SMTP> EHLO localhost
fetchmail: SMTP< 250-snark.thyrsus.com
           Hello localhost [127.0.0.1], pleased to meet you
fetchmail: SMTP< 250-ENHANCEDSTATUSCODES
fetchmail: SMTP< 250-8BITMIME
fetchmail: SMTP< 250-SIZE
fetchmail: SMTP> MAIL FROM:<mutt-dev-owner@mutt.org> SIZE=8328
fetchmail: SMTP< 250 2.1.0 <mutt-dev-owner@mutt.org>... Sender ok
fetchmail: SMTP> RCPT TO:<esr@localhost>
```

```
fetchmail: SMTP< 250 2.1.5 <esr@localhost>... Recipient ok
fetchmail: SMTP> DATA
fetchmail: SMTP< 354 Enter mail, end with "." on a line by itself
#
fetchmail: IMAP< )
fetchmail: IMAP< A0006 OK FETCH completed
fetchmail: IMAP> A0007 FETCH 2 BODY.PEEK[TEXT]
fetchmail: IMAP< * 2 FETCH (BODY[TEXT] {6742}
 (6742 body octets) *********************.**************************.
*******************************.************************.***********
**********.***********************.***************
fetchmail: IMAP< )
fetchmail: IMAP< A0007 OK FETCH completed
fetchmail: SMTP>. (EOM)
fetchmail: SMTP< 250 2.0.0 gB9ffWo08245 Message accepted for delivery
 flushed
fetchmail: IMAP> A0008 STORE 2 +FLAGS (\Seen \Deleted)
fetchmail: IMAP< * 2 FETCH (FLAGS (\Recent \Seen \Deleted))
fetchmail: IMAP< A0008 OK STORE completed
fetchmail: IMAP> A0009 EXPUNGE
fetchmail: IMAP< * 2 EXPUNGE
fetchmail: IMAP< * 1 EXISTS
fetchmail: IMAP< * 0 RECENT
fetchmail: IMAP< A0009 OK Expunged 1 messages
fetchmail: IMAP> A0010 LOGOUT
fetchmail: IMAP< * BYE hurkle IMAP4rev1 server terminating connection
fetchmail: IMAP< A0010 OK LOGOUT completed
fetchmail: 6.1.0 querying hurkle.thyrsus.com (protocol IMAP)
           at Mon, 09 Dec 2002 08:41:42 -0500: poll completed
fetchmail: SMTP> QUIT
fetchmail: SMTP< 221 2.0.0 snark.thyrsus.com closing connection
fetchmail: normal termination, status 0
```

The `-v` option makes what *fetchmail* is doing discoverable (by letting you see the protocol exchanges). This is *immensely* useful. I considered it so important that I wrote special code to mask account passwords out of `-v` transaction dumps so that they could be passed around and posted without anyone having to remember to edit sensitive information out of them.

This turned out to be a good call. At least eight out of ten problems reported get diagnosed within seconds of a knowledgeable person's eyes seeing a session transcript. There are several

knowledgeable people on the fetchmail mailing list — in fact, because most bugs are easy to diagnose, I seldom have to handle them myself.

Over the years, *fetchmail* has acquired a reputation as a rather bulletproof program. It can be misconfigured, but it very seldom outright breaks. Betting that this has nothing to do with the fact that the exact circumstances of eight out of ten bugs are rapidly discoverable would not be smart.

We can learn from this example. The lesson is this: Don't let your debugging tools be mere afterthoughts or treat them as throwaways. They are your windows into the code; don't just knock crude holes in the walls, finish and glaze them. If you plan to keep the code maintained, you're always going to need to let light into it.

Case Study: GCC

GCC, the GNU C compiler used on most modern Unixes, is perhaps an even better example of engineering for transparency. GCC is organized as a sequence of processing stages knit together by a driver program. The stages are: preprocessor, parser, code generator, assembler, and linker.

Each of the first three stages takes in a readable textual format and emits a readable textual format (the assembler has to emit and the linker to accept binary formats, pretty much by definition). With various command-line options of the gcc(1) driver, you can see not just the results after C preprocessing, after assembly generation, and after object code generation — but you can also monitor the results of many intermediate steps in parsing and code generation.

> This is exactly the structure of cc, the first (PDP-11) C compiler.
>
> —
>
> <author>KenThompson</author>

There are many benefits of this organization. One that is particularly important for GCC is regression testing.[60] Because most of the various intermediate formats are textual, deviations from expected results in a regression test are easily spotted and analyzed using simple textual diff operations on the intermediate results; there is no need for specialist dump-analysis tools that may well harbor their own bugs, and in any case would represent an additional maintenance burden.

[60] Regression testing is a method for detecting bugs introduced as software is modified. It consists of periodically checking the output of the changing software for some fixed test input against a snapshot of output captured at an earlier stage of the process and known (or assumed) to be correct.

The design pattern to extract from this example is that the driver program has monitoring switches that merely (but sufficiently) expose the textual data flows among the components. As with *fetchmail's* `-v` option, these options are not afterthoughts; they are designed in for discoverability.

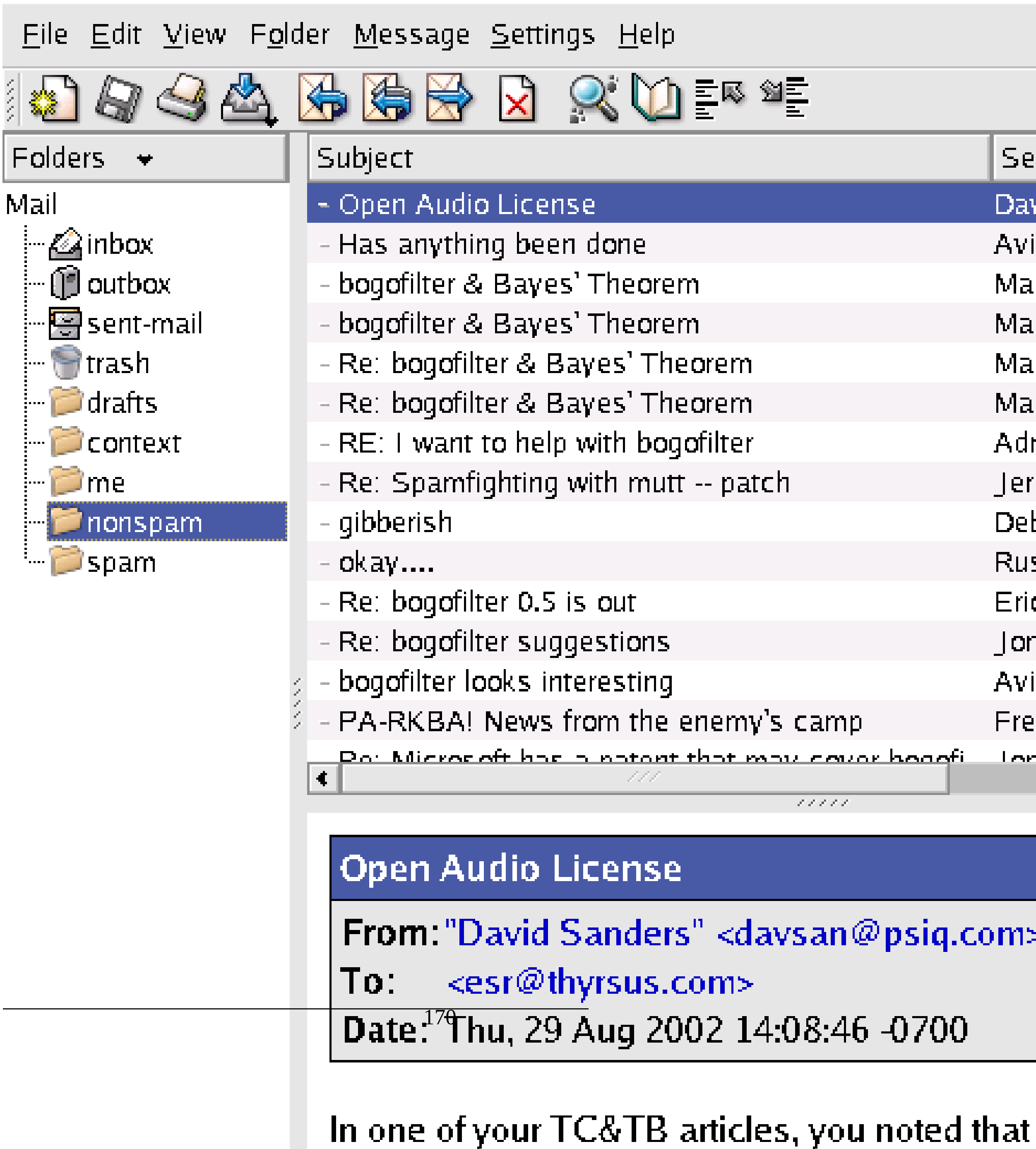
Case Study: *kmail*

kmail is the GUI mailreader distributed with the KDE environment. The *kmail* UI is tastefully and well designed, with many good features including automatic display of enclosed images in a MIME multipart and support for PGP key encryption/decryption. It is friendly to end-users — my beloved but nontechie wife uses and enjoys it.

Many mail user agents make one gesture in the direction of discoverability by having a command that toggles display of all the mail headers, as opposed to a select few like From and Subject. The UI of *kmail* takes this a long step further.

A running *kmail* displays status notifications in a one-line subwindow at the bottom of its window, in small type over a steel-gray background clearly modeled on the Netscape/Mozilla status bar. When you open a mailbox, for example, the status bar displays counts of total and unread messages. The visual presentation is unobtrusive; it is easy to ignore the notifications, but also easy to focus on them if you want to.

Figure 6.2. Screen shot of *kmail*.



The *kmail* GUI is good user-interface design. It's informative, but not distracting; it gets around the reason we adduce in Chapter 11 that the best policy for Unix tools operating normally is usually silence. The authors showed excellent taste in borrowing the look and feel of the browser status bar.

But the extent of the *kmail* developers' tastefulness will not become clear until you have to troubleshoot an installation that is having trouble sending mail. If you watch closely during the send, you will observe that each line of the SMTP transaction with the remote mail transport is echoed into the *kmail* status bar as it happens.

The *kmail* developers neatly avoid a trap that often makes GUI programs like *kmail* a terrible pain in a troubleshooter's fundament. Most design teams with kmail's objectives would have suppressed those messages entirely, fearing that they would give Aunt Tillie a touch of the vapors that would drive her back to the meretricious pseudo-simplicity of a Windows box.

Instead, they designed for transparency — they made the transaction messages show, but also made them visually easy to ignore. By getting the presentation right, they managed to please both Aunt Tillie and her geeky nephew Melvin who fixes her computer problems. This was brilliant; it's a technique other GUI interfaces could and should emulate.

Ultimately, of course, the visibility of those messages is good for Aunt Tillie, because they mean Melvin is far less likely to throw up his hands in frustration while trying to solve her email problems.

The lesson here is clear. Dumbing down your UI is only the half-smart thing to do. The really smart thing is to find a way to leave the details accessible, but make them unobtrusive.

Case Study: SNG

The program *sng* translates between PNG format and an all-text representation of it (SNG or Scriptable Network Graphics format) that can be examined and modified with an ordinary text editor. Run on a PNG file, it produces an SNG file; run on an SNG file, it recovers the equivalent PNG. The transformation is 100% faithful and lossless in both directions.

In syntactic style, SNG resembles CSS (Cascading Style Sheets), another language for controlling presentation of graphics; this makes at least a gesture in the direction of the Rule of Least Surprise. Here is a test example:

Example 6.2. An SNG Example.

```
#SNG: This is a synthetic SNG test file

# Our first test is a paletted (type 3) image.
IHDR: {
      width: 16;
      height: 19;
      bitdepth: 8;
      using color: palette;
      with interlace;
}

# Sample bit depth chunk
sBIT: {
 red: 8;
 green: 8;
 blue: 8;
}

# An example palette: three colors, one of which
# we will render transparent
PLTE: {
  (0,    0, 255)
  (255,   0,   0)
  "dark slate gray",
}

# Suggested palette
sPLT {
  name: "A random suggested palette";
  depth: 8;
  (0,    0, 255), 255, 7;
  (255,   0,   0), 255, 5;
  ( 70,  70,  70), 255, 3;
}

# The viewer will actually use this...
IMAGE: {
```

```
  pixels base64
2222222222222222
2222222222222222
0000001111100000
0000011111110000
0000111001111000
0001110000111100
0001110000111100
0000110001111000
0000000011110000
0000000111100000
0000001111000000
0000001111000000
0000000000000000
0000000110000000
0000001111000000
0000001111000000
0000000110000000
2222222222222222
2222222222222222
}

tEXt: {                          # Ordinary text chunk
 keyword: "Title";
 text: "Sample SNG script";
}

# Test file ends here
```

The point of this tool is to enable users to edit various obscure PNG chunk types that are not necessarily supported by conventional graphics editors. Rather than writing special-purpose code to grovel through the PNG binary format, the user can simply flip an image into an all-text representation, edit that, and massage it back. Another potential application is in making images amenable to version control; under most version-control systems, text files are much easier to manage than binary blobs, and diff operations on SNG representations actually have some possibility of yielding useful information.

The gains here go beyond the time not spent writing special-purpose code for manipulating binary PNGs, however. The code of the *sng* program itself is not especially transparent, but it promotes transparency in larger systems of programs by making the entire contents of PNGs discoverable.

Case Study: The Terminfo Database

The terminfo database is a collection of descriptions of video-display terminals. Each entry describes the escape sequences that perform various manipulations on the terminal screen, such as inserting or deleting lines, erasing from the cursor position to end of line or screen, or beginning and ending screen highlights such as reverse video, underline, or blink.

The terminfo database is primarily used by the curses(3) libraries. These underlie the “roguelike” interface style we discuss in Chapter 11, and some very widely used programs such as mutt(1), lynx(1), and slrn(1). Though the terminal emulators such as xterm(1) that run on today’s bitmapped displays all have capabilities that are minor variations on those of the ANSI X3.64 standard and the venerable VT100 terminal, there is still enough variation that hardwiring ANSI capabilities into applications would be a bad idea. Terminfo is also worth studying because problems that are logically similar to the one it addressed arise constantly in managing other kinds of peripheral hardware that doesn’t have a standard way to report their own capabilities.

The design of terminfo benefits from experience with an earlier capability format called termcap. The database of termcap descriptions lived in a textual format in one big file, `/etc/termcap`; though this format is now obsolete, your Unix system almost certainly includes a copy.

Normally, the key used to look up your terminal type entry is the environment variable `TERM`, which for purposes of this case study is set by magic.[61] Applications that use terminfo (or termcap) pay a small penalty in startup lag; when the curses(3) library initializes itself, it has to look up the entry corresponding to `TERM` and load the entry into memory.

Experience with termcap showed that the startup penalty was dominated by the time required to parse the textual representation of capabilities. Accordingly, terminfo entries are binary structure dumps that can be marshaled and unmarshaled more quickly. There is a master textual format for the entire database, the terminfo capability file. That file (or individual entries) can be compiled to binary form with the terminfo compiler tic(1); binary entries can be decompiled to the editable text format by infocmp(1).

The design superficially contradicts the advice we gave in Chapter 5 against binary caches, but this is actually the extreme case in which that’s a good tactic. Edits to the text masters are very rare — in fact, Unixes normally ship with the terminfo database precompiled and the text master serving

[61] Actually, `TERM` is set by the system at login time. For actual terminals on serial lines, the mapping from tty lines to `TERM` values is set from a system configuration file at boot time; the details vary among Unixes. Terminal emulators like xterm(1) set this variable themselves.

primarily as documentation. Thus, the synchronization and inconsistency problems that would normally militate against this approach almost never arise.

The designers of terminfo could have optimized for speed in a second way. The entire database of binary entries could have been put in some kind of big opaque database file. What they actually did instead was more clever and more in the Unix spirit. Terminfo entries live in a directory hierarchy, usually on modern Unixes under `/usr/share/terminfo`. Consult the terminfo(5) man page to find the location on your system.

If you look in the terminfo directory, you'll see subdirectories named by single printable characters. Under each of these are the entries for each terminal type that has a name beginning with that letter. The goal of this organization was to avoid having to do a linear search of a very large directory; under more modern Unix file systems, which represent directories with B-trees or other structures optimized for fast lookup, the subdirectories won't be necessary.

> I found that even on a fairly modern Unix, splitting a big directory up into subdirectories can improve performance substantially. It was tens of thousands of files, an authorized-user database for a big educational institution, on a late-model DEC Alpha running DEC's Unix. (Subdirectories named by first and last letter of name — e.g., "johnson" would be in directory "j_n" — worked best of the schemes we tested. Using the first two letters wasn't nearly as good, because there were a lot of systematically-generated names which differed only toward the end.) This may just say that sophisticated directory indexing is still not as common as it should be... but even so, that makes an organization which works well without it more portable than one which requires it.
>
> —
>
> <author>HenrySpencer</author>

Thus, the cost of opening a terminfo entry is two file system lookups and a file open. But since mining the same entry from one big database would have required a lookup and open for the database, the incremental cost for terminfo's organization is at most one file system lookup. Actually, it's less than that; it's the cost difference between one file system lookup and whatever retrieval method the one big database would have used. This is probably marginal, and quite tolerable once per application at startup time.

Terminfo uses the file system itself as a simple hierarchical database. This is a superb bit of constructive laziness, obeying the Rule of Economy and the Rule of Transparency. It means that all the ordinary tools for navigating, examining and modifying the file system can be used to navigate,

examine, and modify the terminfo database; no special ones (other than tic(1) and infocmp(1) for packing and unpacking the individual records) need to be written and debugged. It also means that work on speeding up database access would be work on speeding up the file system itself, tuning that would benefit many more applications than just users of curses(3).

There is one additional advantage of this organization that doesn't come up in the terminfo case; you get to use Unix's permissions mechanism rather than having to invent your own access-control layer with its own bugs. This falls out as a consequence of adopting the "everything is a file" philosophy of Unix rather than trying to fight it.

The terminfo directory layout is rather space-inefficient on most Unix file systems. The entries are usually between 400 and 1400 bytes long, but file systems normally allocate a minimum of 4K for every nonempty disk file. The designers accepted this cost for the same reason they chose a packed binary format, to cut the startup latency of terminfo-using programs to a minimum. Disk capacity for constant price has exploded over a thousandfold since, tending to vindicate that decision.

The contrast with the formats used by the Microsoft Windows registry files is instructive. Registries are property databases used by both Windows itself and applications. Each registry lives in one big file. Registries contain a mix of text and binary data that requires specialized editing tools. The one-big-file approach leads, among other things, to the notorious 'registry creep' phenomenon; average access time rises without bound as new entries are added. Because there is no standard API for editing the registry provided by the system, applications use ad-hoc code to edit it themselves, making it notoriously subject to corruption that can lock up the entire system.

Using the Unix file system as a database is a tactic other applications with simple database requirements might do well to emulate. Good reasons not to do it are more likely to have to do with the database keys not naturally looking like filenames than they are with any performance problems. In any case, it's the sort of good fast hack that can be very useful in prototyping.

Case Study: Freeciv Data Files

Freeciv is an open-source strategy game inspired by Sid Meier's classic *Civilization II*. In it, each player begins with a wandering band of neolithic nomads and builds a civilization. Player civilizations may explore and colonize the world, fight wars, engage in trade, and research technological advances. Some players may actually be artificial intelligences; solitaire play against these can be challenging. One wins either by conquering the world or by being the first player to reach a technology level sufficient to get a starship to Alpha Centauri. Sources and documentation are available at the project site [http://www.freeciv.org/].

Figure 6.3. Main window of a Freeciv game.



In Chapter 7 we'll exhibit the Freeciv strategy game as an example of client-server partitioning, with the server maintaining shared state and the client concentrating on GUI presentation. But this game has another notable architectural feature; much of the game's fixed data, rather than being wired into the server code, is expressed in a property registry read in by the game server at startup time.

The game's registry files are written in a textual data-file format that assembles text strings (with associated text and numeric properties) into various internal lists of important data (such as nations and unit types) in the game server. The minilanguage has an include directive, so game data can be broken up into semantic units (different files) that are each separately editable. This design choice has been carried through to such an extent that it's possible to define new nations and new unit types simply by creating new declarations in the data files, without touching the server code at all.

The Freeciv server's startup parsing has an interesting feature that creates something of a conflict between two of Unix's design rules, and is therefore worth closer examination. The server ignores property names it doesn't know how to use. This makes it possible to declare properties that the server doesn't yet use without breaking the startup parsing. It means that development of the game data (policy) and the server engine (mechanism) can be cleanly separated. On the other hand, it also means startup parsing won't catch simple misspellings of attribute names. This quiet failure seems to violate the Rule of Repair.

To resolve this conflict, notice that it's the server's job to *use* the registry data, but the task of carefully error-checking that data could be handed off to another program to be run by human editors each time the registry is modified. One Unix solution would be a separate auditing program that analyzes either a machine-readable specification of the ruleset format or the source of the server code to determine the set of properties it uses, parses the Freeciv registry to determine the set of properties it provides, and prepares a difference report.[62]

The aggregate of all Freeciv data files is functionally similar to a Windows registry, and even uses a syntax resembling the textual portions of registries. But the creep and corruption problems we noted with the Windows registry don't crop up here because no program (either within or outside the Freeciv suite) *writes* to these files. It's a read-only registry edited only by the game's maintainers.

The performance impact of data-file parsing is minimized because for each file the operation is performed only once, at either client or server startup time.

[62] The ur-ancestor of such validator programs under Unix was *lint*, a validator for C code separate from the C compiler. Though GCC has absorbed its functions, old Unix hands are still apt to refer to the process of running a validator as 'linting', and the name survives in utilities such as *xmllint*.

Designing for Transparency and Discoverability

To design for transparency and discoverability, you need to apply every tactic for keeping your code simple, and also concentrate on the ways in which your code is a communication to other human beings. The first questions to ask, after “Will this design work?” are “Will it be readable to other people? Is it elegant?” We hope it is clear by now that these questions are not fluff and that elegance is not a luxury. These qualities in the human reaction to software are essential for reducing its bugginess and increasing its long-term maintainability.

The Zen of Transparency

One pattern that emerges from the examples we’ve examined so far in this chapter is this: If you want transparent code, the most effective route is simply not to layer too much abstraction over what you are manipulating with the code.

In Chapter 4’s section on the value of detachment, our advice was to abstract and simplify and generalize, to try and detach from the particular, accidental conditions under which a design problem was posed. The advice to abstract does not actually contradict the advice against excessive abstractions we’re developing here, because there is a difference between getting free of assumptions and forgetting the problem you’re trying to solve. This is part of what we were driving at when we developed the idea that glue layers need to be kept thin.

One of the main lessons of Zen is that we ordinarily see the world through a haze of preconceptions and fixed ideas that proceed from our desires. To achieve enlightenment, we must follow the Zen teaching not merely to let go of desire and attachment, but to experience reality exactly as it is — without the preconceptions and the fixed ideas getting in the way.

This is excellent pragmatic advice for software designers. It’s part of what’s implicit in the classic Unix advice to be minimalist. Software designers are clever people who form ideas (abstractions) about the application domains they deal with. They organize the software they write around those ideas. Then, when debugging, they often find they have great trouble seeing through those ideas to what is actually going on.

Any Zen master would recognize this problem instantly, yell “Three pounds of flax!”, and probably clout the student a good one.[63] Consciously designing for transparency is a slightly less mystical way of addressing it.

[63] See the koan called *Tozan’s Three Pounds* in the *Gateless Gate* [Mumon].

In Chapter 4 we criticized object-oriented programming in terms likely to prove a bit shocking to programmers who were raised on the 1990s gospel of OO. Object-oriented design doesn't have to be over-complicated design, but we've observed that too often it is. Too many OO designs are spaghetti-like tangles of is-a and has-a relationships, or feature thick layers of glue in which many of the objects seem to exist simply to hold places in a steep-sided pyramid of abstractions. Such designs are the opposite of transparent; they are (notoriously) opaque and difficult to debug.

As we've previously noted, Unix programmers are the original zealots about modularity, but tend to go about it in a quieter way. Keeping glue layers thin is part of it; more generally, our tradition teaches us to build lower, hugging the ground with algorithms and structures that are designed to be simple and transparent.

As with Zen art, the simplicity of good Unix code depends on exacting self-discipline and a high level of craft, neither of which are necessarily apparent on casual inspection. Transparency is hard work, but worth the effort for more than merely artistic reasons. Unlike Zen art, software requires debugging — and usually needs continuing maintenance, forward-porting, and adaptation throughout its lifetime. Transparency is therefore more than an esthetic triumph; it is a victory that will be reflected in lower costs throughout the software's life cycle.

Coding for Transparency and Discoverability

Transparency and discoverability, like modularity, are primarily properties of designs, not code. It is not sufficient to get right the low-level elements of style, such as indenting code in a clear and consistent way or having good variable-naming conventions. These qualities have much more to do with code properties that are less obvious to inspection. Here are a few to think about:

- What is the maximum static depth of your procedure-call hierarchy? That is, leaving out recursions, how many levels of call might a human have to model mentally to understand the operation of the code? Hint: If it's more than four, beware.
- Does the code have invariant properties[64] that are both strong and visible? Invariant properties help human beings reason about code and detect problem cases.

[64] An invariant is a property of a software design that is preserved by every operation in it. For example, in most databases it is an invariant that no two records may have the same key. In a C program that correctly manipulates strings, every string buffer must contain a terminating NUL byte on exit from each string function. In an inventory system, no parts count can hold a number less than zero.

- Are the function calls in your APIs individually orthogonal, or do they have too many magic flags and mode bits that have a single call doing multiple tasks? Avoiding mode flags entirely can lead to a cluttered API with too many nigh-identical functions, but the obverse error (lots of easily-forgotten and confusable mode flags) is even more common.
- Are there a handful of prominent data structures or a single global scoreboard that captures the high-level state of the system? Is this state easy to visualize and inspect, or is it diffused among many individual global variables or objects that are hard to find?
- Is there a clean, one-to-one mapping between data structures or classes in your program and the entities in the world that they represent?
- Is it easy to find the portion of the code responsible for any given function? How much attention have you paid to the readability not just of individual functions and modules but of the whole codebase?
- Does the code proliferate special cases or avoid them? Every special case could interact with every other special case; all those potential collisions are bugs waiting to happen. But even more importantly, special cases make the code harder to understand.
- How many magic numbers (unexplained constants) does the code have in it? Is it easy to discover the implementation's limits (such as critical buffer sizes) by inspection?

It's best for code to be simple. But if it answers these sorts of questions well, it can be very complex without putting an impossible cognitive burden on a human maintainer.

The reader might find it instructive to compare these with our checklist questions about modularity in Chapter 4.

Transparency and Avoiding Overprotectiveness

Close kin to the programmer tendency to build overelaborate castles of abstractions is a tendency to overprotect others from the low-level details. While it's not bad practice to hide those details in the program's normal mode of operation (*fetchmail*'s `-v` switch is off by default), they should be discoverable. There's an important difference between hiding them and making them inaccessible.

Programs that *cannot* reveal what they are doing make troubleshooting far more difficult. Thus, experienced Unix users actually take the presence of debugging and instrumentation switches as a

good sign, and their absence as possibly a bad one. Absence suggests an inexperienced or careless developer; presence suggests one with enough wisdom to follow the Rule of Transparency.

The temptation to overprotect is especially strong in GUI applications targeted for end users, like mail readers. One reason Unix developers have been cool toward GUI interfaces is that, in their designers' haste to make them 'user-friendly' each one often becomes frustratingly opaque to anyone who has to solve user problems — or, indeed, interact with it anywhere outside the narrow range predicted by the user-interface designer.

Worse, programs that are opaque about what they are doing tend to have a lot of assumptions baked into them, and to be frustrating or brittle or both in any use case not anticipated by the designer. Tools that look glossy but shatter under stress are not good long-term value.

Unix tradition pushes for programs that are flexible for a broader range of uses and troubleshooting situations, including the ability to present as much state and activity information to the user as the user indicates he is willing to handle. This is good for troubleshooting; it is also good for growing smarter, more self-reliant users.

Transparency and Editable Representations

Another theme that emerges from these examples is the value of programs that flip a problem out of a domain in which transparency is hard into one in which it is easy. Audacity, sng(1) and the tic(1)/infocmp(1) pair all have this property. The objects they manipulate are not readily conformable to the hand and eye; audio files are not visual objects, and although images expressed in PNG format are visual, the complexities of PNG annotation chunks are not. All three applications turn manipulation of their binary file formats into a problem to which human beings can more readily apply intuition and competences gained from everyday experience.

A rule all these examples follow is that they degrade the representation as little as possible — in fact, they translate it reversibly and losslessly. This property is very important, and worth implementing even if there is no obvious application demand for that kind of 100% fidelity. It gives potential users confidence that they can experiment without degrading their data.

All the advantages of textual data-file formats that we discussed in Chapter 5 also apply to the textual formats that sng(1), infocmp(1) and their kin generate. One important application for sng(1) is robotic generation of PNG image annotations by scripts — because sng(1) exists, such scripts are easier to write.

Whenever you face a design problem that involves editing some kind of complex binary object, the Unix tradition encourages asking first off whether you can write a tool analogous to sng(1) or the tic(1)/infocmp(1) pair that can do a lossless mapping to an editable textual format and back. There is no established term for programs of this kind, but we'll call them *textualizers*.

If the binary object is dynamically generated or very large, then it may not be practical or possible to capture all the state with a textualizer. In that case, the equivalent task is to write a browser. The paradigm example is fsdb(1), the file-system debugger supported under various Unixes; there is a Linux equivalent called debugfs(1). The psql(1) used to browse PostgreSQL databases, and the smbclient(1) program that can be used to query Windows file shares on a SAMBA-equipped Linux machine, are two more. All five are simple CLI programs that could be driven by scripts and test harnesses.

Writing a textualizer or browser is a valuable exercise for at least four reasons:

- *You gain an excellent learning experience.* There may be other ways that are as good to learn about the structure of the object, but none that are obviously better.
- *You gain the ability to dump the contents of the structure for inspection and debugging.* Because such a tool makes dumping easy, you'll do it more. You'll get more information, probably leading to more insight.
- *You gain the ability to easily generate test loads and unusual cases.* This means you are more likely to probe the odd corners of the object's state space — and to break the associated software, so you can fix it before your users break it.
- *You gain code you may be able to reuse.* If you're careful about how you write the browser/textualizer and keep the CLI interpreter properly separated from the marshaling/unmarshaling library, you may find you have code that can be reused for your actual application.

After you've done this, you may well discover that it's possible to apply the "separated engine and interface" pattern (see Chapter 11) using your textualizer/debugger as the engine. All the usual benefits of this pattern will apply.

> It is desirable, although often difficult, for a textualizer to be able to read and write even a damaged binary object. For one thing, it lets you generate damaged test cases to stress-test software; for another, it can make emergency repairs a whole lot easier. It may be hard to handle cases in which the *structure* of the object is messed up, but at least you should handle cases in which the *content* of the structure is nonsense, e.g., by showing nonsense values in hex and converting the hex back to the values.
>
> —
> <author>HenrySpencer</author>

Transparency, Fault Diagnosis, and Fault Recovery

Yet another benefit of transparency, related to ease of debugging, is that transparent systems are easier to perform recovery actions on after a bug bites — and, often, more resistant to damage from bugs in the first place.

In comparing the terminfo database with Windows registries we noted that registries are notoriously subject to being corrupted by buggy application code. This can make the entire system unusable. Even if it doesn't, recovery can be difficult if the corruption confuses the specialized registry-editing tools.

Our Unix case studies illustrate ways that designing for transparency can prevent this class of problem. Because the terminfo database is not one big file, botching one terminfo entry does not make the whole terminfo data set unusable. Fully textual one-big-file formats like termcap are usually parsed with methods which (unlike block reads of binary structure dumps) can recover from single-point errors. Syntax errors in an SNG file can be corrected by hand without requiring specialized editors that might refuse to load a damaged PNG image.

Going back to the *kmail* case study, that program makes fault diagnosis easier because it obeys the Rule of Repair: SMTP failures are noisy, usefully so. You don't have to decode a layer of obfuscatory messages generated by *kmail* itself to see what the interaction with the SMTP server looks like. All you have to do is look in the right place, because *kmail* is being transparent and not

throwing away information about the error state. (It helps that SMTP itself is textual and includes human-readable status messages in its transactions.)

Discoverability tools like textualizers and browsers also make fault diagnosis easier. We've already touched on one reason: they make inspecting the state of the system easier. But there is another effect at work as well; textualized versions of data tend to have useful redundancies (such as using whitespace for visual separation as well as explicit delimiters for parsing). These are present to make them easier to read for humans, but also have the effect of making them more resistant to being irreparably trashed by point failures. A corrupted chunk in a PNG file is seldom recoverable, but the human capacity for pattern recognition and reasoning from context might be able to repair the equivalent SNG form.

Over and over again, the Rule of Robustness is clear. Simplicity plus transparency lowers costs, reduces everybody's stress, and frees people to concentrate on new problems rather than cleaning up after old mistakes.

Designing for Maintainability

Software is maintainable to the extent that people who are not its author can successfully understand and modify it. Maintainability demands more than code that works; it demands code that follows the Rule of Clarity and communicates successfully to human beings as well as the computer.

Unix programmers have a lot of implicit knowledge available to them about what makes for maintainable code, because Unix hosts source code that goes back decades. For reasons we'll discuss in Chapter 17, Unix programmers learn a tendency to scrap and rebuild rather than patching grubby code (see Rob Pike's meditation on this subject in Chapter 1). Thus, any sources that have survived more than a decade of evolutionary pressure have been selected for maintainability. These old, successful, well-established projects with maintainable code are the community's models for practice.

A question Unix programmers — and especially Unix programmers in the open-source world — learn to ask about tools they are evaluating for use is: "Is this code live, dormant, or dead?" Live code has an active developer community attached to it. Dormant code has often become dormant because the pain of maintaining it exceeded its utility to its originators. Dead code has been dormant for so long that it would be easier to reimplement an equivalent from scratch. If you want your code to live, investing effort to make it maintainable (and therefore attractive to future maintainers) will be one of the most effective ways you can spend your time.

Code that is designed to be both transparent and discoverable has gone a long way toward being maintainable. But there are other practices we can observe in the model projects in this chapter that are worth emulating.

One very important practice is an application of the Rule of Clarity: choosing simple algorithms. In Chapter 1 we quoted Ken Thompson: “When in doubt, use brute force”. Thompson understood the full cost of complicated algorithms — not just that they’re more bug-prone when initially implemented, but that they’re harder for maintainers down the line to understand.

Another important practice is the inclusion of hacker’s guides. It has always been highly approved behavior for source code distributions to include guide documents informally describing the key data structures and algorithms in the code. In fact, Unix programmers have often been better about producing hacker’s guides than they are about writing end-user documentation.

The open-source community has seized on and elaborated this custom. Besides being advice to future maintainers, hacker’s guides for open-source projects are also designed to make it easy for casual contributors to add features or fix bugs. The Design Notes file shipped with *fetchmail* is representative. The Linux kernel sources include literally dozens of these.

In Chapter 19 we’ll describe conventions that Unix developers have evolved for making source code distributions easy to examine and easy to build running code from. These practices, too, promote maintainability.

Chapter 7. Multiprogramming

Separating Processes to Separate Function

If we believe in data structures, we must believe in independent (hence simultaneous) processing. For why else would we collect items within a structure? Why do we tolerate languages that give us the one without the other?

--

<author>AlanPerlis</author>

Epigrams in Programming, in ACM SIGPLAN (Vol 17 #9, 1982)

The most characteristic program-modularization technique of Unix is splitting large programs into multiple cooperating processes. This has usually been called ‘multiprocessing’ in the Unix world, but in this book we revive the older term ‘multiprogramming’ to avoid confusion with multiprocessor hardware implementations.

Multiprogramming is a particularly murky area of design, one in which there are few guidelines to good practice. Many programmers with excellent judgment about how to break up code into subroutines nevertheless wind up writing whole applications as monster single-process monoliths that founder on their own internal complexity.

The Unix style of design applies the do-one-thing-well approach at the level of cooperating programs as well as cooperating routines within a program, emphasizing small programs connected by well-defined interprocess communication or by shared files. Accordingly, the Unix operating system encourages us to break our programs into simpler subprocesses, and to concentrate on the interfaces between these subprocesses. It does this in at least three fundamental ways:

- by making process-spawning cheap;
- by providing methods (shellouts, I/O redirection, pipes, message-passing, and sockets) that make it relatively easy for processes to communicate;
- by encouraging the use of simple, transparent, textual data formats that can be passed through pipes and sockets.

Inexpensive process-spawning and easy process control are critical enablers for the Unix style of programming. On an operating system such as VAX VMS, where starting processes is expensive and slow and requires special privileges, one must build monster monoliths because one has no choice. Fortunately the trend in the Unix family has been toward lower fork(2) overhead rather than higher. Linux, in particular, is famously efficient this way, with a process-spawn faster than thread-spawning on many other operating systems.[65]

Historically, many Unix programmers have been encouraged to think in terms of multiple cooperating processes by experience with shell programming. Shell makes it relatively easy to set up groups of multiple processes connected by pipes, running either in background or foreground or a mix of the two.

In the remainder of this chapter, we'll look at the implications of cheap process-spawning and discuss how and when to apply pipes, sockets, and other interprocess communication (IPC) methods to partition your design into cooperating processes. (In the next chapter, we'll apply the same separation-of-functions philosophy to interface design.)

While the benefit of breaking programs up into cooperating processes is a reduction in global complexity, the cost is that we have to pay more attention to the design of the protocols which are used to pass information and commands between processes. (In software systems of all kinds, bugs collect at interfaces.)

In Chapter 5 we looked at the lower level of this design problem — how to lay out application protocols that are transparent, flexible and extensible. But there is a second, higher level to the problem which we blithely ignored. That is the problem of designing state machines for each side of the communication.

It is not hard to apply good style to the syntax of application protocols, given models like SMTP or BEEP or XML-RPC. The real challenge is not protocol syntax but protocol *logic*—designing a protocol that is both sufficiently expressive and deadlock-free. Almost as importantly, the protocol has to be *seen* to be expressive and deadlock-free; human beings attempting to model the behavior of the communicating programs in their heads and verify its correctness must be able to do so.

In our discussion, therefore, we will focus on the kinds of protocol logic one naturally uses with each kind of interprocess communication.

[65] See, for example, the results quoted in *Improving Context Switching Performance of Idle Tasks under Linux* [Appleton].

Separating Complexity Control from Performance Tuning

First, though, we need to dispose of a few red herrings. Our discussion is *not* going to be about using concurrency to improve performance. Putting that concern before developing a clean architecture that minimizes global complexity is premature optimization, the root of all evil (see Chapter 12 for further discussion).

A closely related red herring is threads (that is, multiple concurrent processes sharing the same memory-address space). Threading is a performance hack. To avoid a long diversion here, we'll examine threads in more detail at the end of this chapter; the summary is that they do not reduce global complexity but rather *increase* it, and should therefore be avoided save under dire necessity.

Respecting the Rule of Modularity, on the other hand, is *not* a red herring; doing so can make your programs — and your life — simpler. All the reasons for process partitioning are continuous with the reasons for module partitioning that we developed in Chapter 4.

Another important reason for breaking up programs into cooperating processes is for better security. Under Unix, programs that must be run by ordinary users, but must have write access to security-critical system resources, get that access through a feature called the *setuid bit*.[66] Executable files are the smallest unit of code that can hold a setuid bit; thus, every line of code in a setuid executable must be trusted. (Well-written setuid programs, however, take all necessary privileged actions first and then drop their privileges back to user level for the remainder of their existence.)

Usually a setuid program only needs its privileges for one or a small handful of operations. It is often possible to break up such a program into cooperating processes, a smaller one that needs setuid and a larger one that does not. When we can do this, only the code in the smaller program has to be trusted. It is in significant part because this kind of partitioning and delegation is possible that Unix has a better security track record[67] than its competitors.

Taxonomy of Unix IPC Methods

[66]A setuid program runs not with the privileges of the user calling it, but with the privileges of the owner of the executable. This feature can be used to give restricted, program-controlled access to things like the password file that nonadministrators should not be allowed to modify directly.

[67]That is, a better record measured in security breaches per total machine hours of Internet exposure.

As in single-process program architectures, the simplest organization is the best. The remainder of this chapter will present IPC techniques roughly in order of escalating complexity of programming them. Before using a later, more complex technique, you should prove by demonstration — with prototypes and benchmark results — that no earlier and simpler technique will do. Often you will surprise yourself.

Handing off Tasks to Specialist Programs

In the simplest form of interprogram cooperation enabled by inexpensive process spawning, a program runs another to accomplish a specialized task. Because the called program is often specified as a Unix shell command through the system(3) call, this is often called *shelling out* to the called program. The called program inherits the user's keyboard and display and runs to completion. When it exits, the calling program resumes control of the keyboard and display and resumes execution.[68] Because the calling program does not communicate with the called program during the callee's execution, protocol design is not an issue in this kind of cooperation, except in the trivial sense that the caller may pass command-line arguments to the callee to change its behavior.

The classic Unix case of shelling out is calling an editor from within a mail or news program. In the Unix tradition one does *not* bundle purpose-built editors into programs that require general text-edited input. Instead, one allows the user to specify an editor of his or her choice to be called when editing needs to be done.

The specialist program usually communicates with its parent through the file system, by reading or modifying file(s) with specified location(s); this is how editor or mailer shellouts work.

In a common variant of this pattern, the specialist program may accept input on its standard input, and be called with the C library entry point `popen(..., "w")` or as part of a shellscript. Or it may send output to its standard output, and be called with `popen(..., "r")` or as part of a shellscript. (If it both reads from standard input and writes to standard output, it does so in a batch mode, completing all reads before doing any writes.) This kind of child process is not usually referred to as a shellout; there is no standard jargon for it, but it might well be called a 'bolt-on'.

They key point about all these cases is that the specialist programs don't handshake with the parent while they are running. They have an associated protocol only in the trivial sense that whichever program (master or slave) is accepting input from the other has to be able to parse it.

[68] A common error in programming shellouts is to forget to block signals in the parent while the subprocess runs. Without this precaution, an interrupt typed to the subprocess can have unwanted side effects on the parent process.

Case Study: The *mutt* Mail User Agent

The *mutt* mail user agent is the modern representative of the most important design tradition in Unix email programs. It has a simple screen-oriented interface with single-keystroke commands for browsing and reading mail.

When you use *mutt* as a mail composer (either by calling it with an address as a command-line argument or by using one of the reply commands), it examines the process environment variable `EDITOR`, and then generates a temporary file name. The value of the `EDITOR` variable is called as a command with the tempfile name as an argument.[69] When that command terminates, *mutt* resumes on the assumption that the temporary file contains the desired mail text.

Almost all Unix mail- and netnews-composition programs observe the same convention. Because they do, composer implementers don't need to write a hundred inevitably diverging editors, and users don't need to learn a hundred divergent interfaces. Instead, users can carry their chosen editors with them.

An important variant of this strategy shells out to a small proxy program that passes the specialist job to an already-running instance of a big program, like an editor or a Web browser. Thus, developers who normally have an instance of *emacs* running on their X display can set **EDITOR=emacsclient**, and have a buffer pop open in their *emacs* when they request editing in *mutt*. The point of this is not really to save memory or other resources, it's to enable the user to unify all editing in a single *emacs* process (so that, for example, cut and paste among buffers can carry along internal *emacs* state information like font highlighting).

Pipes, Redirection, and Filters

After Ken Thompson and Dennis Ritchie, the single most important formative figure of early Unix was probably Doug McIlroy. His invention of the *pipe* construct reverberated through the design of Unix, encouraging its nascent do-one-thing-well philosophy and inspiring most of the later forms of IPC in the Unix design (in particular, the socket abstraction used for networking).

Pipes depend on the convention that every program has initially available to it (at least) two I/O data streams: standard input and standard output (numeric file descriptors 0 and 1 respectively). Many programs can be written as *filters*, which read sequentially from standard input and write only to standard output.

[69] Actually, the above is a slight oversimplification. See the discussion of `EDITOR` and `VISUAL` in Chapter 10 for the rest of the story.

Normally these streams are connected to the user's keyboard and display, respectively. But Unix shells universally support *redirection* operations which connect these standard input and output streams to files. Thus, typing

```
ls >foo
```

sends the output of the directory lister ls(1) to a file named 'foo'. On the other hand, typing:

```
wc <foo
```

causes the word-count utility wc(1) to take its standard input from the file 'foo', and deliver a character/word/line count to standard output.

The pipe operation connects the standard output of one program to the standard input of another. A chain of programs connected in this way is called a *pipeline*. If we write

```
ls | wc
```

we'll see a character/word/line count for the current directory listing. (In this case, only the line count is really likely to be useful.)

> One favorite pipeline was "**bc | speak**"—a talking desk calculator. It knew number names up to a vigintillion.
>
> —
>
> <author>DougMcIlroy</author>

It's important to note that all the stages in a pipeline run concurrently. Each stage waits for input on the output of the previous one, but no stage has to exit before the next can run. This property will be important later on when we look at interactive uses of pipelines, like sending the lengthy output of a command to more(1).

It's easy to underestimate the power of combining pipes and redirection. As an instructive example, *The Unix Shell As a 4GL* [Schaffer-Wolf] shows that with these facilities as a framework, a handful of

simple utilities can be combined to support creating and manipulating relational databases expressed as simple textual tables.

The major weakness of pipes is that they are unidirectional. It's not possible for a pipeline component to pass control information back up the pipe other than by terminating (in which case the previous stage will get a `SIGPIPE` signal on the next write). Accordingly, the protocol for passing data is simply the receiver's input format.

So far, we have discussed anonymous pipes created by the shell. There is a variant called a *named pipe* which is a special kind of file. If two programs open the file, one for reading and the other for writing, a named pipe acts like a pipe-fitting between them. Named pipes are a bit of a historical relic; they have been largely displaced from use by named *sockets*, which we'll discuss below. (For more on the history of this relic, see the discussion of System V IPC below.)

Case Study: Piping to a Pager

Pipelines have many uses. For one example, Unix's process lister ps(1) lists processes to standard output without caring that a long listing might scroll off the top of the user's display too quickly for the user to see it. Unix has another program, more(1), which displays its standard input in screen-sized chunks, prompting for a user keystroke after displaying each screenful.

Thus, if the user types "**ps | more**", piping the output of ps(1) to the input of more(1), successive page-sized pieces of the list of processes will be displayed after each keystroke.

The ability to combine programs like this can be extremely useful. But the real win here is not cute combinations; it's that because both pipes and more(1) exist, *other programs can be simpler*. Pipes mean that programs like ls(1) (and other programs that write to standard out) don't have to grow their own pagers — and we're saved from a world of a thousand built-in pagers (each, naturally, with its own divergent look and feel). Code bloat is avoided and global complexity reduced.

As a bonus, if anyone needs to customize pager behavior, it can be done in *one* place, by changing *one* program. Indeed, multiple pagers can exist, and will all be useful with every application that writes to standard output.

In fact, this has actually happened. On modern Unixes, more(1) has been largely replaced by less(1), which adds the capability to scroll back in the displayed file rather than just forward.[70] Because less(1) is decoupled from the programs that use it, it's possible to simply alias 'more' to 'less' in

[70] The less(1) man page explains the name by observing "Less is more".

your shell, set the environment variable `PAGER` to ‘less’ (see Chapter 10), and get all the benefits of a better pager with all properly-written Unix programs.

Case Study: Making Word Lists

A more interesting example is one in which pipelined programs cooperate to do some kind of data transformation for which, in less flexible environments, one would have to write custom code.

Consider the pipeline

```
tr -c '[:alnum:]' '[\n*]' | sort -iu | grep -v '^[0-9]*$'
```

The first command translates non-alphanumerics on standard input to newlines on standard output. The second sorts lines on standard input and writes the sorted data to standard output, discarding all but one copy of spans of adjacent identical lines. The third discards all lines consisting solely of digits. Together, these generate a sorted wordlist to standard output from text on standard input.

Case Study: *pic2graph*

Shell source code for the program pic2graph(1) ships with the *groff* suite of text-formatting tools from the Free Software Foundation. It translates diagrams written in the PIC language to bitmap images. Example 7.1 shows the pipeline at the heart of this code.

Example 7.1. The *pic2graph* pipeline.

```
(echo ".EQ"; echo $eqndelim; echo ".EN"; echo ".PS";cat;echo ".PE")|\
    groff -e -p $groffpic_opts -Tps >${tmp}.ps \
    && convert -crop 0x0 $convert_opts ${tmp}.ps ${tmp}.${format} \
    && cat ${tmp}.${format}
```

The pic2graph(1) implementation illustrates how much one pipeline can do purely by calling preexisting tools. It starts by massaging its input into an appropriate form, continues by feeding it through groff(1) to produce PostScript, and finishes by converting the PostScript to a bitmap. All

these details are hidden from the user, who simply sees PIC source go in one end and a bitmap ready for inclusion in a Web page come out the other.

This is an interesting example because it illustrates how pipes and filtering can adapt programs to unexpected uses. The program that interprets PIC, pic(1), was originally designed only to be used for embedding diagrams in typeset documents. Most of the other programs in the toolchain it was part of are now semiobsolescent. But PIC remains handy for new uses, such as describing diagrams to be embedded in HTML. It gets a renewed lease on life because tools like pic2graph(1) can bundle together all the machinery needed to convert the output of pic(1) into a more modern format.

We'll examine pic(1) more closely, as a minilanguage design, in Chapter 8.

Case Study: bc(1) and dc(1)

Part of the classic Unix toolkit dating back to Version 7 is a pair of calculator programs. The dc(1) program is a simple calculator that accepts text lines consisting of reverse-Polish notation (RPN) on standard input and emits calculated answers to standard output. The bc(1) program accepts a more elaborate infix syntax resembling conventional algebraic notation; it includes as well the ability to set and read variables and define functions for elaborate formulas.

While the modern GNU implementation of bc(1) is standalone, the classic version passed commands to dc(1) over a pipe. In this division of labor, bc(1) does variable substitution and function expansion and translates infix notation into reverse-Polish — but doesn't actually do calculation itself, instead passing RPN translations of input expressions to dc(1) for evaluation.

There are clear advantages to this separation of function. It means that users get to choose their preferred notation, but the logic for arbitrary-precision numeric calculation (which is moderately tricky) does not have to be duplicated. Each of the pair of programs can be less complex than one calculator with a choice of notations would be. The two components can be debugged and mentally modeled independently of each other.

In Chapter 8 we will reexamine these programs from a slightly different example, as examples of domain-specific minilanguages.

Anti-Case Study: Why Isn't *fetchmail* a Pipeline?

In Unix terms, *fetchmail* is an uncomfortably large program that bristles with options. Thinking about the way mail transport works, one might think it would be possible to decompose it into a

pipeline. Suppose for a moment it were broken up into several programs: a couple of fetch programs to get mail from POP3 and IMAP sites, and a local SMTP injector. The pipeline could pass Unix mailbox format. The present elaborate *fetchmail* configuration could be replaced by a shellscript containing command lines. One could even insert filters in the pipeline to block spam.

```
#!/bin/sh
imap jrandom@imap.ccil.org | spamblocker | smtp jrandom
imap jrandom@imap.netaxs.com | smtp jrandom
# pop ed@pop.tems.com | smtp jrandom
```

This would be very elegant and Unixy. Unfortunately, it can't work. We touched on the reason earlier; pipelines are unidirectional.

One of the things the fetcher program (*imap* or *pop*) would have to do is decide whether to send a delete request for each message it fetches. In *fetchmail*'s present organization, it can delay sending that request to the POP or IMAP server until it knows that the local SMTP listener has accepted responsibility for the message. The pipelined, small-component version would lose that property.

Consider, for example, what would happen if the *smtp* injector fails because the SMTP listener reports a disk-full condition. If the fetcher has already deleted the mail, we lose. This means the fetcher cannot delete mail until it is notified to do so by the *smtp* injector. This in turn raises a host of questions. How would they communicate? What message, exactly, would the injector pass back? The global complexity of the resulting system, and its vulnerability to subtle bugs, would almost certainly be higher than that of a monolithic program.

Pipelines are a marvelous tool, but not a universal one.

Wrappers

The opposite of a shellout is a *wrapper*. A wrapper creates a new interface for a called program, or specializes it. Often, wrappers are used to hide the details of elaborate shell pipelines. We'll discuss interface wrappers in Chapter 11. Most specialization wrappers are quite simple, but nevertheless very useful.

As with shellouts, there is no associated protocol because the programs do not communicate during the execution of the callee; but the wrapper usually exists to specify arguments that modify the callee's behavior.

Case Study: Backup Scripts

Specialization wrappers are a classic use of the Unix shell and other scripting languages. One kind of specialization wrapper that is both common and representative is a backup script. It may be a one-liner as simple as this:

```
tar -czvf /dev/st0 "$@"
```

This is a wrapper for the tar(1) tape archiver utility which simply supplies one fixed argument (the tape device `/dev/st0`) and passes to tar all the other arguments supplied by the user ("`$@`").[71]

Security Wrappers and Bernstein Chaining

One common use of wrapper scripts is as *security wrappers*. A security script may call a gatekeeper program to check some sort of credential, then conditionally execute another based on the status value returned by the gatekeeper.

Bernstein chaining is a specialized security-wrapper technique first invented by Daniel J. Bernstein, who has employed it in a number of his packages. (A similar pattern appears in commands like nohup(1) and su(1), but the conditionality is absent.) Conceptually, a Bernstein chain is like a pipeline, but each successive stage replaces the previous one rather than running concurrently with it.

The usual application is to confine security-privileged applications to some sort of gatekeeper program, which can then hand state to a less privileged one. The technique pastes several programs together using execs, or possibly a combination of forks and execs. The programs are all named on one command line. Each program performs some function and (if successful) runs exec(2) on the rest of its command line.

Bernstein's *rblsmtpd* package is a prototypical example. It serves to look up a host in the antispam DNS zone of the Mail Abuse Prevention System. It does this by doing a DNS query on the IP address passed into it in the `TCPREMOTEIP` environment variable. If the query is successful, then *rblsmtpd* runs its own SMTP that discards the mail. Otherwise the remaining command-line arguments are presumed to constitute a mail transport agent that knows the SMTP protocol, and are handed to exec(2) to be run.

[71] A common error is to use `$*` rather than "`$@`". This does bad things when handed a filename with embedded spaces.

Another example can be found in Bernstein's *qmail* package. It contains a program called *condredirect*. The first parameter is an email address, and the remainder a gatekeeper program and arguments. *condredirect* forks and execs the gatekeeper with its arguments. If the gatekeeper exits successfully, *condredirect* forwards the email pending on stdin to the specified email address. In this case, opposite to that of *rblsmtpd*, the security decision is made by the child; this case is a bit more like a classical shellout.

A more elaborate example is the *qmail* POP3 server. It consists of three programs, *qmail-popup*, *checkpassword*, and *qmail-pop3d*. *Checkpassword* comes from a separate package cleverly called *checkpassword*, and unsurprisingly it checks the password. The POP3 protocol has an authentication phase and mailbox phase; once you enter the mailbox phase you cannot go back to the authentication phase. This is a perfect application for Bernstein chaining.

The first parameter of *qmail-popup* is the hostname to use in the POP3 prompts. The rest of its parameters are forked and passed to exec(2), after the POP3 username and password have been fetched. If the program returns failure, the password must be wrong, so *qmail-popup* reports that and waits for a different password. Otherwise, the program is presumed to have finished the POP3 conversation, so *qmail-popup* exits.

The program named on *qmail-popup*'s command line is expected to read three null-terminated strings from file descriptor 3.[72] These are the username, password, and response to a cryptographic challenge, if any. This time it's *checkpassword* which accepts as parameters the name of *qmail-pop3d* and its parameters. The *checkpassword* program exits with failure if the password does not match; otherwise it changes to the user's uid, gid, and home directory, and executes the rest of its command line on behalf of that user.

Bernstein chaining is useful for situations in which the application needs setuid or setgid privileges to initialize a connection, or to acquire some credential, and then drop those privileges so that following code does not have to be trusted. Following the exec, the child program cannot set its real user ID back to root. It's also more flexible than a single process, because you can modify the behavior of the system by inserting another program into the chain.

For example, *rblsmtpd* (mentioned above) can be inserted into a Bernstein chain, in between tcpserver (from the *ucspi-tcp* package) and the real SMTP server, typically *qmail-smtpd*. However, it works with inetd(8) and **sendmail -bs** as well.

[72] *qmail-popup*'s standard input and standard output are the socket, and standard error (which will be file descriptor 2) goes to a log file. File descriptor 3 is guaranteed to be the next to be allocated. As an infamous kernel comment once observed: "You are not expected to understand this".

Slave Processes

Occasionally, child programs both accept data from and return data to their callers through pipes connected to standard input and output, interactively. Unlike simple shellouts and what we have called 'bolt-ons' above, both master and slave processes need to have internal state machines to handle a protocol between them without deadlocking or racing. This is a drastically more complex and more difficult-to-debug organization than a simple shellout.

Unix's popen(3) call can set up either an input pipe or an output pipe for a shellout, but not both for a slave process — this seems intended to encourage simpler programming. And, in fact, interactive master-slave communication is tricky enough that it is normally only used when either (a) the implied protocol is utterly trivial, or (b) the slave process has been designed to speak an application protocol along the lines we discussed in Chapter 5. We'll return to this issue, and ways to cope with it, in Chapter 8.

When writing a master/slave pair, it is good practice for the master to support a command-line switch or environment variable that allows callers to set their own slave command. Among other things, this is useful for debugging; you will often find it handy during development to invoke the real slave process from within a harness that monitors and logs transactions between slave and master.

If you find that master/slave interactions in your program are becoming nontrivial, it may be time to think about going the rest of the way to a more peer-to-peer organization, using techniques like sockets or shared memory.

Case Study: *scp* and *ssh*

One common case in which the implied protocol really is trivial is progress meters. The scp(1) secure-copy command calls ssh(1) as a slave process, intercepting enough information from ssh's standard output to reformat the reports as an ASCII animation of a progress bar.[73]

Peer-to-Peer Inter-Process Communication

All the communication methods we've discussed so far have a sort of implicit hierarchy about them, with one program effectively controlling or driving another and zero or limited feedback passing in the opposite direction. In communications and networking we frequently need channels that are *peer-to-peer*, usually (but not necessarily) with data flowing freely in both directions. We'll

[73]The friend who suggested this case study comments: "Yes, you can get away with this technique...if there are just a few easily-recognizable nuggets of information coming back from the slave process, and you have tongs and a radiation suit".

survey peer-to-peer communications methods under Unix here, and develop some case studies in later chapters.

Tempfiles

The use of tempfiles as communications drops between cooperating programs is the oldest IPC technique there is. Despite drawbacks, it's still useful in shellscripts, and in one-off programs where a more elaborate and coordinated method of communication would be overkill.

The most obvious problem with using tempfiles as an IPC technique is that it tends to leave garbage lying around if processing is interrupted before the tempfile can be deleted. A less obvious risk is that of collisions between multiple instances of a program using the same name for a tempfile. This is why it is conventional for shellscripts that make tempfiles to include $$ in their names; this shell variable expands to the process-ID of the enclosing shell and effectively guarantees that the filename will be unique (the same trick is supported in Perl).

Finally, if an attacker knows the location to which a tempfile will be written, it can overwrite on that name and possibly either read the producer's data or spoof the consumer process by inserting modified or spurious data into the file.[74] This is a security risk. If the processes involved have root privileges, this is a very serious risk. It can be mitigated by setting the permissions on the tempfile directory carefully, but such arrangements are notoriously likely to spring leaks.

All these problems aside, tempfiles still have a niche because they're easy to set up, they're flexible, and they're less vulnerable to deadlocks or race conditions than more elaborate methods. And sometimes, nothing else will do. The calling conventions of your child process may require that it be handed a file to operate on. Our first example of a shellout to an editor demonstrates this perfectly.

Signals

The simplest and crudest way for two processes on the same machine to communicate with each other is for one to send the other a *signal*. Unix signals are a form of soft interrupt; each one has a default effect on the receiving process (usually to kill it). A process can declare a *signal handler* that overrides the default action for the signal; the handler is a function that is executed asynchronously when the signal is received.

[74] A particularly nasty variant of this attack is to drop in a named Unix-domain socket where the producer and consumer programs are expecting the tempfile to be.

Signals were originally designed into Unix as a way for the operating system to notify programs of certain errors and critical events, not as an IPC facility. The `SIGHUP` signal, for example, is sent to every program started from a given terminal session when that session is terminated. The `SIGINT` signal is sent to whatever process is currently attached to the keyboard when the user enters the currently-defined interrupt character (often control-C). Nevertheless, signals can be useful for some IPC situations (and the POSIX-standard signal set includes two signals, `SIGUSR1` and `SIGUSR2`, intended for this use). They are often employed as a control channel for *daemons* (programs that run constantly, invisibly, in background), a way for an operator or another program to tell a daemon that it needs to either reinitialize itself, wake up to do work, or write internal-state/debugging information to a known location.

> I insisted `SIGUSR1` and `SIGUSR2` be invented for BSD. People were grabbing system signals to mean what they needed them to mean for IPC, so that (for example) some programs that segfaulted would not coredump because `SIGSEGV` had been hijacked.
>
> This is a general principle — people will want to hijack any tools you build, so you have to design them to either be un-hijackable or to be hijacked cleanly. Those are your only choices. Except, of course, for being ignored—a highly reliable way to remain unsullied, but less satisfying than might at first appear.
>
> —
>
> <author>KenArnold</author>

A technique often used with signal IPC is the so-called *pidfile*. Programs that will need to be signaled will write a small file to a known location (often in `/var/run` or the invoking user's home directory) containing their process ID or PID. Other programs can read that file to discover that PID. The pidfile may also function as an implicit *lock file* in cases where no more than one instance of the daemon should be running simultaneously.

There are actually two different flavors of signals. In the older implementations (notably V7, System III, and early System V), the handler for a given signal is reset to the default for that signal whenever the handler fires. The result of sending two of the same signal in quick succession is therefore usually to kill the process, no matter what handler was set.

The BSD 4.x versions of Unix changed to “reliable” signals, which do not reset unless the user explicitly requests it. They also introduced primitives to block or temporarily suspend processing of a given set of signals. Modern Unixes support both styles. You should use the BSD-style

nonresetting entry points for new code, but program defensively in case your code is ever ported to an implementation that does not support them.

Receiving N signals does not necessarily invoke the signal handler N times. Under the older System V signal model, two or more signals spaced very closely together (that is, within a single timeslice of the target process) can result in various race conditions[75] or anomalies. Depending on what variant of signals semantics the system supports, the second and later instances may be ignored, may cause an unexpected process kill, or may have their delivery delayed until earlier instances have been processed (on modern Unixes the last is most likely).

The modern signals API is portable across all recent Unix versions, but not to Windows or classic (pre-OS X) MacOS.

System Daemons and Conventional Signals

Many well-known system daemons accept `SIGHUP` (originally the signal sent to programs on a serial-line drop, such as was produced by hanging up a modem connection) as a signal to reinitialize (that is, reload their configuration files); examples include Apache and the Linux implementations of bootpd(8), gated(8), inetd(8), mountd(8), named(8), nfsd(8), and ypbind(8). In a few cases, `SIGHUP` is accepted in its original sense of a session-shutdown signal (notably in Linux pppd(8)), but that role nowadays generally goes to `SIGTERM`.

`SIGTERM` (‘terminate’) is often accepted as a graceful-shutdown signal (this is as distinct from `SIGKILL`, which does an immediate process kill and cannot be blocked or handled). `SIGTERM` actions often involve cleaning up tempfiles, flushing final updates out to databases, and the like.

When writing daemons, follow the Rule of Least Surprise: use these conventions, and read the manual pages to look for existing models.

Case Study: *fetchmail’s* Use of Signals

The *fetchmail* utility is normally set up to run as a daemon in background, periodically collecting mail from all remote sites defined in its run-control file and passing the mail to the local SMTP listener on port 25 without user intervention. *fetchmail* sleeps for a user-defined interval (defaulting to 15 minutes) between collection attempts, so as to avoid constantly loading the network.

[75] A ‘race condition’ is a class of problem in which correct behavior of the system relies on two independent events happening in the right order, but there is no mechanism for ensuring that they actually will. Race conditions produce intermittent, timing-dependent problems that can be devilishly difficult to debug.

When you invoke **fetchmail** with no arguments, it checks to see if you have a *fetchmail* daemon already running (it does this by looking for a pidfile). If no daemon is running, *fetchmail* starts up normally using whatever control information has been specified in its run-control file. If a daemon is running, on the other hand, the new *fetchmail* instance just signals the old one to wake up and collect mail immediately; then the new instance terminates. In addition, **fetchmail -q** sends a termination signal to any running *fetchmail* daemon.

Thus, typing **fetchmail** means, in effect, “poll now and leave a daemon running to poll later; don’t bother me with the detail of whether a daemon was already running or not”. Observe that the detail of which particular signals are used for wakeup and termination is something the user doesn’t have to know.

Sockets

Sockets were developed in the BSD lineage of Unix as a way to encapsulate access to data networks. Two programs communicating over a socket typically see a bidirectional byte stream (there are other socket modes and transmission methods, but they are of only minor importance). The byte stream is both sequenced (that is, even single bytes will be received in the same order sent) and reliable (socket users are guaranteed that the underlying network will do error detection and retry to ensure delivery). Socket descriptors, once obtained, behave essentially like file descriptors.

> Sockets differ from read/write in one important case. If the bytes you send arrive, but the receiving machine fails to ACK, the sending machine’s TCP/IP stack will time out. So getting an error does *not* necessarily mean that the bytes didn’t arrive; the receiver may be using them. This problem has profound consequences for the design of reliable protocols, because you have to be able to work properly when you don’t know what was received in the past. Local I/O is ‘yes/no’. Socket I/O is ‘yes/no/maybe’. And nothing can ensure delivery — the remote machine might have been destroyed by a comet.
>
> —
>
> <author>KenArnold</author>

At the time a socket is created, you specify a *protocol family* which tells the network layer how the name of the socket is interpreted. Sockets are usually thought of in connection with the Internet, as a way of passing data between programs running on different hosts; this is the AF_INET socket family, in which addresses are interpreted as host-address and service-number pairs. However, the AF_UNIX (aka AF_LOCAL) protocol family supports the same socket abstraction for communication between two processes on the same machine (names are

interpreted as the locations of special files analogous to bidirectional named pipes). As an example, client programs and servers using the X windowing system typically use AF_LOCAL sockets to communicate.

All modern Unixes support BSD-style sockets, and as a matter of design they are usually the right thing to use for bidirectional IPC no matter where your cooperating processes are located. Performance pressure may push you to use shared memory or tempfiles or other techniques that make stronger locality assumptions, but under modern conditions it is best to assume that your code will need to be scaled up to distributed operation. More importantly, those locality assumptions may mean that portions of your system get chummier with each others' internals than ought to be the case in a good design. The separation of address spaces that sockets enforce is a feature, not a bug.

To use sockets gracefully, in the Unix tradition, start by designing an *application protocol* for use between them — a set of requests and responses which expresses the semantics of what your programs will be communicating about in a succinct way. We've already discussed the some major issues in the design of application protocols in Chapter 5.

Sockets are supported in all recent Unixes, under Windows, and under classic MacOS as well.

Case Study: PostgreSQL

PostgreSQL is an open-source database program. Had it been implemented as a monster monolith, it would be a single program with an interactive interface that manipulates database files on disk directly. Interface would be welded together with implementation, and two instances of the program attempting to manipulate the same database at the same time would have serious contention and locking issues.

Instead, the PostgreSQL suite includes a server called *postmaster* and at least three client applications. One *postmaster* server process per machine runs in background and has exclusive access to the database files. It accepts requests in the SQL query minilanguage through TCP/IP sockets, and returns answers in a textual format as well. When the user runs a PostgreSQL client, that client opens a session to *postmaster* and does SQL transactions with it. The server can handle several client sessions at once, and sequences requests so that they don't interfere with each other.

Because the front end and back end are separate, the server doesn't need to know anything except how to interpret SQL requests from a client and send SQL reports back to it. The clients, on

the other hand, don't need to know anything about how the database is stored. Clients can be specialized for different needs and have different user interfaces.

This organization is quite typical for Unix databases — so much so that it is often possible to mix and match SQL clients and SQL servers. The interoperability issues are the SQL server's TCP/IP port number, and whether client and server support the same dialect of SQL.

Case Study: Freeciv

In Chapter 6, we introduced Freeciv as an example of transparent data formats. But more critical to the way it supports multiplayer gaming is the client/server partitioning of the code. This is a representative example of a program in which the application needs to be distributed over a wide-area network and handles communication through TCP/IP sockets.

The state of a running Freeciv game is maintained by a server process, the game engine. Players run GUI clients which exchange information and commands with the server through a packet protocol. All game logic is handled in the server. The details of GUI are handled in the client; different clients support different interface styles.

This is a very typical organization for a multiplayer online game. The packet protocol uses TCP/IP as a transport, so one server can handle clients running on different Internet hosts. Other games that are more like real-time simulations (notably first-person shooters) use raw Internet datagram protocol (UDP) and trade lower latency for some uncertainty about whether any given packet will be delivered. In such games, users tend to be issuing control actions continuously, so sporadic dropouts are tolerable, but lag is fatal.

Shared Memory

Whereas two processes using sockets to communicate may live on different machines (and, in fact, be separated by an Internet connection spanning half the globe), shared memory requires producers and consumers to be co-resident on the same hardware. But, if your communicating processes can get access to the same physical memory, shared memory will be the fastest way to pass information between them.

Shared memory may be disguised under different APIs, but on modern Unixes the implementation normally depends on the use of mmap(2) to map files into memory that can be shared between processes. POSIX defines a shm_open(3) facility with an API that supports using files as shared

memory; this is mostly a hint to the operating system that it need not flush the pseudofile data to disk.

Because access to shared memory is not automatically serialized by a discipline resembling read and write calls, programs doing the sharing must handle contention and deadlock issues themselves, typically by using semaphore variables located in the shared segment. The issues here resemble those in multithreading (see the end of this chapter for discussion) but are more manageable because default is *not* to share memory. Thus, problems are better contained.

On systems where it is available and reliable, the Apache web server's scoreboard facility uses shared memory for communication between an Apache master process and the load-sharing pool of Apache images that it manages. Modern X implementations also use shared memory, to pass large images between client and server when they are resident on the same machine, to avoid the overhead of socket communication. Both uses are performance hacks justified by experience and testing, rather than being architectural choices.

The mmap(2) call is supported under all modern Unixes, including Linux and the open-source BSD versions; this is described in the Single Unix Specification. It will not normally be available under Windows, MacOS classic, and other operating systems.

Before purpose-built mmap(2) was available, a common way for two processes to communicate was for them to open the same file, and then delete that file. The file wouldn't go away until all open filehandles were closed, but some old Unixes took the link count falling to zero as a hint that they could stop updating the on-disk copy of the file. The downside was that your backing store was the file system rather than a swap device, the file system the deleted file lived on couldn't be unmounted until the programs using it closed, and attaching new processes to an existing shared memory segment faked up in this way was tricky at best.

After Version 7 and the split between the BSD and System V lineages, the evolution of Unix interprocess communication took two different directions. The BSD direction led to sockets. The AT&T lineage, on the other hand, developed named pipes (as previously discussed) and an IPC facility, specifically designed for passing binary data and based on shared-memory bidirectional message queues. This is called 'System V IPC'—or, among old timers, 'Indian Hill' IPC after the AT&T facility where it was first written.

The upper, message-passing layer of System V IPC has largely fallen out of use. The lower layer, which consists of shared memory and semaphores, still has significant applications under circumstances in which one needs to do mutual-exclusion locking and some global data sharing

among processes running on the same machine. These System V shared memory facilities evolved into the POSIX shared-memory API, supported under Linux, the BSDs, MacOS X and Windows, but not classic MacOS.

By using these shared-memory and semaphore facilities (shmget(2), semget(2), and friends) one can avoid the overhead of copying data through the network stack. Large commercial databases (including Oracle, DB2, Sybase, and Informix) use this technique heavily.

Problems and Methods to Avoid

While BSD-style sockets over TCP/IP have become the dominant IPC method under Unix, there are still live controversies over the right way to partition by multiprogramming. Some obsolete methods have not yet completely died, and some techniques of questionable utility have been imported from other operating systems (often in association with graphics or GUI programming). We'll be touring some dangerous swamps here; beware the crocodiles.

Obsolescent Unix IPC Methods

Unix (born 1969) long predates TCP/IP (born 1980) and the ubiquitous networking of the 1990s and later. Anonymous pipes, redirection, and shellout have been in Unix since very early days, but the history of Unix is littered with the corpses of APIs tied to obsolescent IPC and networking models, beginning with the `mx()` facility that appeared in Version 6 (1976) and was dropped before Version 7 (1979).

Eventually BSD sockets won out as IPC was unified with networking. But this didn't happen until after fifteen years of experimentation that left a number of relics behind. It's useful to know about these because there are likely to be references to them in your Unix documentation that might give the misleading impression that they're still in use. These obsolete methods are described in more detail in *Unix Network Programming* [Stevens90].

> The real explanation for all the dead IPC facilities in old AT&T Unixes was politics. The Unix Support Group was headed by a low-level manager, while some projects that used Unix were headed by vice presidents. They had ways to make irresistible requests, and would not brook the objection that most IPC mechanisms are interchangeable.
>
> —
>
> <author>DougMcIlroy</author>

System V IPC

The System V IPC facilities are message-passing facilities based on the System V shared memory facility we described earlier.

Programs that cooperate using System V IPC usually define shared protocols based on exchanging short (up to 8K) binary messages. The relevant manual pages are msgctl(2) and friends. As this style has been largely superseded by text protocols passed between sockets, we do not give an example here.

The System V IPC facilities are present in Linux and other modern Unixes. However, as they are a legacy feature, they are not exercised very often. The Linux version is still known to have bugs as of mid-2003. Nobody seems to care enough to fix them.

Streams

Streams networking was invented for Unix Version 8 (1985) by Dennis Ritchie. A re-implementation called STREAMS (yes, it is all-capitals in the documentation) first became available in the 3.0 release of System V Unix (1986). The STREAMS facility provided a full-duplex interface (functionally not unlike a BSD socket, and like sockets, accessible through normal read(2) and write(2) operations after initial setup) between a user process and a specified device driver in the kernel. The device driver might be hardware such as a serial or network card, or it might be a software-only pseudodevice set up to pass data between user processes.

An interesting feature of both streams and STREAMS[76] is that it is possible to push protocol-translation modules into the kernel's processing path, so that the device the user process 'sees' through the full-duplex channel is actually filtered. This capability could be used, for example, to implement a line-editing protocol for a terminal device. Or one could implement protocols such as IP or TCP without wiring them directly into the kernel.

Streams originated as an attempt to clean up a messy feature of the kernel called 'line disciplines' — alternative modes of processing character streams coming from serial terminals and early local-area networks. But as serial terminals faded from view, Ethernet LANs became ubiquitous, and TCP/IP drove out other protocol stacks and migrated into Unix kernels, the extra flexibility provided by STREAMS had less and less utility. In 2003, System V Unix still supports STREAMS, as do some System V/BSD hybrids such as Digital Unix and Sun Microsystems' Solaris.

[76]STREAMS was much more complex. Dennis Ritchie is reputed to have said "Streams means something different when shouted".

Linux and other open-source Unixes have effectively discarded STREAMS. Linux kernel modules and libraries are available from the LiS [http://www.gcom.com/home/linux/lis/] project, but (as of mid-2003) are not integrated into the stock Linux kernel. They will not be supported under non-Unix operating systems.

Remote Procedure Calls

Despite occasional exceptions such as NFS (Network File System) and the GNOME project, attempts to import CORBA, ASN.1, and other forms of remote-procedure-call interface have largely failed — these technologies have not been naturalized into the Unix culture.

There seem to be several underlying reasons for this. One is that RPC interfaces are not readily discoverable; that is, it is difficult to query these interfaces for their capabilities, and difficult to monitor them in action without building single-use tools as complex as the programs being monitored (we examined some of the reasons for this in Chapter 6). They have the same version skew problems as libraries, but those problems are harder to track because they're distributed and not generally obvious at link time.

As a related issue, interfaces that have richer type signatures also tend to be more complex, therefore more brittle. Over time, they tend to succumb to ontology creep as the inventory of types that get passed across interfaces grows steadily larger and the individual types more elaborate. Ontology creep is a problem because structs are more likely to mismatch than strings; if the ontologies of the programs on each side don't exactly match, it can be very hard to teach them to communicate at all, and fiendishly difficult to resolve bugs. The most successful RPC applications, such as the Network File System, are those in which the application domain naturally has only a few simple data types.

The usual argument for RPC is that it permits "richer" interfaces than methods like text streams — that is, interfaces with a more elaborate and application-specific ontology of data types. But the Rule of Simplicity applies! We observed in Chapter 4 that one of the functions of interfaces is as choke points that prevent the implementation details of modules from leaking into each other. Therefore, the main argument in favor of RPC is also an argument that it increases global complexity rather than minimizing it.

With classical RPC, it's too easy to do things in a complicated and obscure way instead of keeping them simple. RPC seems to encourage the production of large, baroque, over-engineered systems with obfuscated interfaces, high global complexity, and serious version-skew and reliability problems — a perfect example of thick glue layers run amok.

Windows COM and DCOM are perhaps the archetypal examples of how bad this can get, but there are plenty of others. Apple abandoned OpenDoc, and both CORBA and the once wildly hyped Java RMI have receded from view in the Unix world as people have gained field experience with them. This may well be because these methods don't actually solve more problems than they cause.

Andrew S. Tanenbaum and Robbert van Renesse have given us a detailed analysis of the general problem in *A Critique of the Remote Procedure Call Paradigm* [Tanenbaum-VanRenesse], a paper which should serve as a strong cautionary note to anyone considering an architecture based on RPC.

All these problems may predict long-term difficulties for the relatively few Unix projects that use RPC. Of these projects, perhaps the best known is the GNOME desktop effort.[77] These problems also contribute to the notorious security vulnerabilities of exposing NFS servers.

Unix tradition, on the other hand, strongly favors transparent and discoverable interfaces. This is one of the forces behind the Unix culture's continuing attachment to IPC through textual protocols. It is often argued that the parsing overhead of textual protocols is a performance problem relative to binary RPCs — but RPC interfaces tend to have latency problems that are far worse, because (a) you can't readily anticipate how much data marshaling and unmarshaling a given call will involve, and (b) the RPC model tends to encourage programmers to treat network transactions as cost-free. Adding even one additional round trip to a transaction interface tends to add enough network latency to swamp any overhead from parsing or marshaling.

Even if text streams were less efficient than RPC, the performance loss would be marginal and linear, the kind better addressed by upgrading your hardware than by expending development time or adding architectural complexity. Anything you might lose in performance by using text streams, you gain back in the ability to design systems that are simpler — easier to monitor, to model, and to understand.

Today, RPC and the Unix attachment to text streams are converging in an interesting way, through protocols like XML-RPC and SOAP. These, being textual and transparent, are more palatable to Unix programmers than the ugly and heavyweight binary serialization formats they replace. While they don't solve all the more general problems pointed out by Tanenbaum and van Renesse, they do in some ways combine the advantages of both text-stream and RPC worlds.

Threads — Threat or Menace?

[77]GNOME's main competitor, KDE, started with CORBA but abandoned it in their 2.0 release. They have been on a quest for lighter-weight IPC methods ever since.

Though Unix developers have long been comfortable with computation by multiple cooperating processes, they do not have a native tradition of using threads (processes that share their entire address spaces). These are a recent import from elsewhere, and the fact that Unix programmers generally dislike them is not merely accident or historical contingency.

From a complexity-control point of view, threads are a bad substitute for lightweight processes with their own address spaces; the idea of threads is native to operating systems with expensive process-spawning and weak IPC facilities.

By definition, though daughter threads of a process typically have separate local-variable stacks, they share the same global memory. The task of managing contentions and critical regions in this shared address space is quite difficult and a fertile source of global complexity and bugs. It can be done, but as the complexity of one's locking regime rises, the chance of races and deadlocks due to unanticipated interactions rises correspondingly.

Threads are a fertile source of bugs because they can too easily know too much about each others' internal states. There is no automatic encapsulation, as there would be between processes with separate address spaces that must do explicit IPC to communicate. Thus, threaded programs suffer from not just ordinary contention problems, but from entire new categories of timing-dependent bugs that are excruciatingly difficult to even reproduce, let alone fix.

Thread developers have been waking up to this problem. Recent thread implementations and standards show an increasing concern with providing thread-local storage, which is intended to limit problems arising from the shared global address space. As threading APIs move in this direction, thread programming starts to look more and more like a controlled use of shared memory.

> Threads often prevent abstraction. In order to prevent deadlock, you often need to know how and if the library you are using uses threads in order to avoid deadlock problems. Similarly, the use of threads in a library could be affected by the use of threads at the application layer.
>
> —
>
> <author>DavidKorn</author>

To add insult to injury, threading has performance costs that erode its advantages over conventional process partitioning. While threading can get rid of some of the overhead of rapidly switching process contexts, locking shared data structures so threads won't step on each other can be just as expensive.

> The X server, able to execute literally millions of ops/second, is *not* threaded; it uses a poll/select loop. Various efforts to make a multithreaded implementation have come to no good result. The costs of locking and unlocking get too high for something as performance-sensitive as graphics servers.
>
> —
>
> <author>JimGettys</author>

This problem is fundamental, and has also been a continuing issue in the design of Unix kernels for symmetric multiprocessing. As your resource-locking gets finer-grained, latency due to locking overhead can increase fast enough to swamp the gains from locking less core memory.

One final difficulty with threads is that threading standards still tend to be weak and underspecified as of mid-2003. Theoretically conforming libraries for Unix standards such as POSIX threads (1003.1c) can nevertheless exhibit alarming differences in behavior across platforms, especially with respect to signals, interactions with other IPC methods, and resource cleanup times. Windows and classic MacOS have native threading models and interrupt facilities quite different from those of Unix and will often require considerable porting effort even for simple threading cases. The upshot is that you cannot count on threaded programs to be portable.

For more discussion and a lucid contrast with event-driven programming, see *Why Threads Are a Bad Idea* [Osterhout96].

Process Partitioning at the Design Level

Now that we have all these methods, what should we do with them?

The first thing to notice is that tempfiles, the more interactive sort of master/slave process relationship, sockets, RPC, and all other methods of bidirectional IPC are at some level equivalent — they're all just ways for programs to exchange data during their lifetimes. Much of what we do in a sophisticated way using sockets or shared memory we could do in a primitive way using tempfiles as mailboxes and signals for notification. The differences are at the edges, in how programs establish communication, where and when one does the marshalling and unmarshalling of messages, in what sorts of buffering problems you may have, and atomicity guarantees you get on the messages (that is, to what extent you can know that the result of a single send action from one side will show up as a single receive event on the other).

We've seen from the PostgreSQL study that one effective way to hold down complexity is to break an application into a client/server pair. The PostgreSQL client and server communicate through an

application protocol over sockets, but very little about the design pattern would change if they used any other bidirectional IPC method.

This kind of partitioning is particularly effective in situations where multiple instances of the application must manage access to resources that are shared among all. A single server process may manage all resource contention, or cooperating peers may each take responsibility for some critical resource.

Client-server partitioning can also help distribute cycle-hungry applications across multiple hosts. Or it may make them suitable for distributed computing across the Internet (as with Freeciv). We'll discuss the related *CLI server* pattern in Chapter 11.

Because all these peer-to-peer IPC techniques are alike at some level, we should evaluate them mainly on the amount of program-complexity overhead they incur, and how much opacity they introduce into our designs. This, ultimately, is why BSD sockets have won over other Unix IPC methods, and why RPC has generally failed to get much traction.

Threads are fundamentally different. Rather than supporting communication among different programs, they support a sort of timesharing within an instance of a single program. Rather than being a way to partition a big program into smaller ones with simpler behavior, threading is strictly a performance hack. It has all the problems normally associated with performance hacks, and a few special ones of its own.

Accordingly, while we should seek ways to break up large programs into simpler cooperating processes, the use of threads within processes should be a last resort rather than a first. Often, you may find you can avoid them. If you can use limited shared memory and semaphores, asynchronous I/O using `SIGIO`, or poll(2)/select(2) rather than threading, do it that way. Keep it simple; use techniques earlier on this list and lower on the complexity scale in preference to later ones.

The combination of threads, remote-procedure-call interfaces, and heavyweight object-oriented design is especially dangerous. Used sparingly and tastefully, any of these techniques can be valuable — but if you are ever invited onto a project that is supposed to feature all three, fleeing in terror might well be an appropriate reaction.

We have previously observed that programming in the real world is all about managing complexity. Tools to manage complexity are good things. But when the effect of those tools is to proliferate complexity rather than to control it, we would be better off throwing them away and starting from zero. An important part of the Unix wisdom is to never forget this.

Chapter 8. Minilanguages

Finding a Notation That Sings

A good notation has a subtlety and suggestiveness which at times makes it almost seem like a live teacher.

--

<author>BertrandRussell</author>

The World of Mathematics (1956)

One of the most consistent results from large-scale studies of error patterns in software is that programmer error rates in defects per hundreds of lines are largely independent of the language in which the programmers are coding.[78] Higher-level languages, which allow you to get more done in fewer lines, mean fewer bugs as well.

Unix has a long tradition of hosting little languages specialized for a particular application domain, languages that can enable you to drastically reduce the line count of your programs. Domain-specific language examples include the numerous Unix typesetting languages (*troff*, *eqn*, *tbl*, *pic*, *grap*), shell utilities (*awk*, *sed*, *dc*, *bc*), and software development tools (*make*, *yacc*, *lex*). There is a fuzzy boundary between domain-specific languages and the more flexible sort of application run-control file (*sendmail*, BIND, X); another with data-file formats; and another with scripting languages (which we'll survey in Chapter 14).

Historically, domain-specific languages of this kind have been called 'little languages' or 'minilanguages' in the Unix world, because early examples were small and low in complexity relative to general-purpose languages (all three terms for the category are in common use). But if the application domain is complex (in that it has lots of different primitive operations or involves manipulation of intricate data structures), an application language for it may have to be rather more complex than some general-purpose languages. We'll keep the traditional term 'minilanguage' to emphasize that the wise course is usually to keep these designs as small and simple as possible.

The domain-specific little language is an extremely powerful design idea. It allows you to define your own higher-level language to specify the appropriate methods, rules, and algorithms for the task at hand, reducing global complexity relative to a design that uses hardwired lower-level code

[78] Les Hatton reports by email on the analysis in his book in preparation, *Software Failure*: "Provided you use executable line counts for the density measure, the injected defect densities vary less between languages than they do between engineers by about a factor of 10".

for the same ends. You can get to a minilanguage design in at least three ways, two of them good and one of them dangerous.

One right way to get there is to realize up front that you can use a minilanguage design to push a given specification of a programming problem up a level, into a notation that is more compact and expressive than you could support in a general-purpose language. As with code generation and data-driven programming, a minilanguage lets you take practical advantage of the fact that the defect rate in your software will be largely independent of the level of the language you are using; more expressive languages mean shorter programs and fewer bugs.

The second right way to get to a minilanguage design is to notice that one of your specification file formats is looking more and more like a minilanguage — that is, it is developing complex structures and implying actions in the application you are controlling. Is it trying to describe control flow as well as data layouts? If so, it may be time to promote that control flow from being implicit to being explicit in your specification language.

The wrong way to get to a minilanguage design is to extend your way to it, one patch and crufty added feature at a time. On this path, your specification file keeps sprouting more implied control flow and more tangled special-purpose structures until it has become an ad-hoc language without your noticing it. Some legendary nightmares have been spawned this way; the example every Unix guru will think of (and shudder over) is the `sendmail.cf` configuration file associated with the *sendmail* mail transport.

Sadly, most people do their first minilanguage the wrong way, and only realize later what a mess it is. Then the question is: how to clean it up? Sometimes the answer implies rethinking the entire application design. Another notorious example of language-by-feature creep was the editor *TECO*, which grew first macros and then loops and conditionals as programmers wanted to use it to package increasingly complex editing routines. The resulting ugliness was eventually fixed by a redesign of the entire editor to be based on an intentional language; this is how Emacs Lisp (which we'll survey below) evolved.

All sufficiently complicated specification files aspire to the condition of minilanguages. Therefore, it will often be the case that your only defense against designing a bad minilanguage is knowing how to design a good one. This need not be a huge step or involve knowing a lot of formal language theory; with modern tools, learning a few relatively simple techniques and bearing good examples in mind as you design should be sufficient.

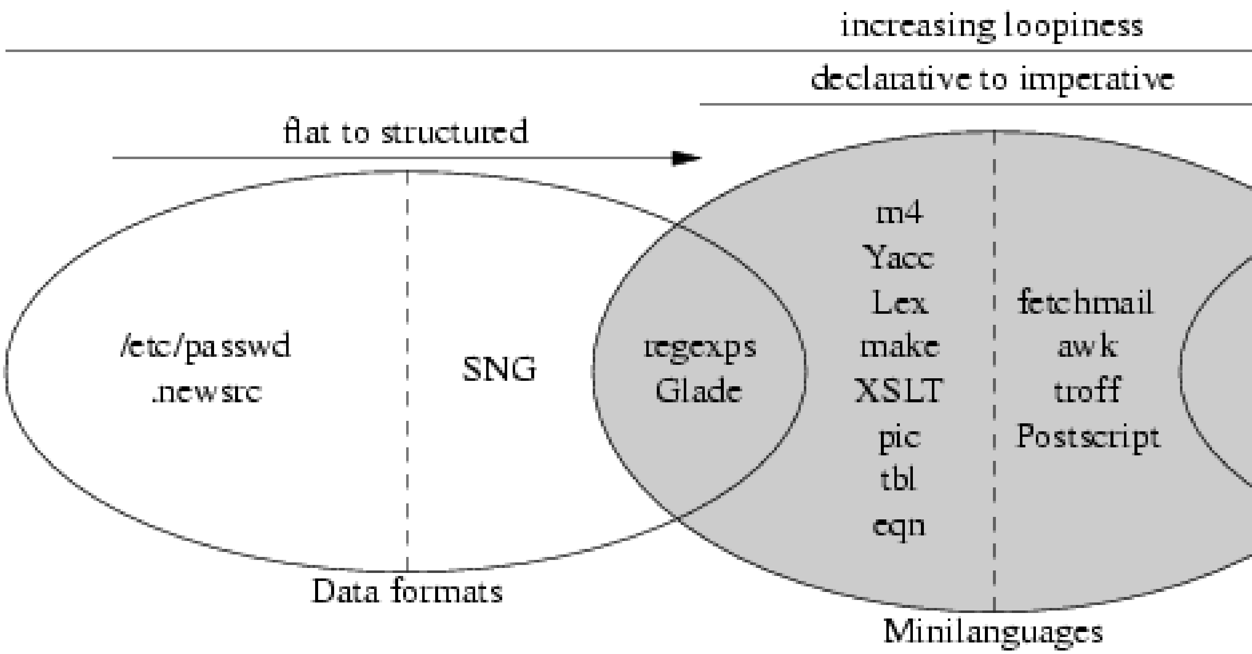
In this chapter we'll examine all the kinds of minilanguages normally supported under Unix, and try to identify the kinds of situation in which each of them represents an effective design solution. This chapter is not meant to be an exhaustive catalog of Unix languages, but rather to bring out the design principles involved in structuring an application around a minilanguage. We'll have much more to say about languages for general-purpose programming in Chapter 14.

We'll need to start by doing a little taxonomy, so we'll know what we're talking about later on.

Understanding the Taxonomy of Languages

All the languages in Figure 8.1 are described in case studies, either in this chapter or elsewhere in this book. For the general-purpose interpreters near the right-hand side, see Chapter 14.

Figure 8.1. Taxonomy of languages.



In Chapter 5 we looked at Unix conventions for data files. There's a spectrum of complexity in these. At the low end are files that make simple associations between names and properties; the `/etc/passwd` and `.newsrc` formats are good examples. Further up the scale we start to get formats that marshal or serialize data structures; the PNG and SNG formats are (equivalent) good examples of this.

A structured data-file format starts to border on being a minilanguage when it expresses not just structure but actions performed on some interpretive context (that is, memory that is outside the data file itself). XML markups tend to straddle this border; the example we'll look at here is *Glade*, a code generator for building GUI interfaces. Formats that are both designed to be read and written by humans (rather than just programs) and are used to generate code, are firmly in the realm of minilanguages. *yacc* and *lex* are the classic examples. We'll discuss *glade*, *yacc* and *lex* in Chapter 9.

The Unix macro processor, *m4*, is another very simple declarative minilanguage (that is, one in which the program is expressed as a set of desired relationships or constraints rather than explicit actions). It has often been used as a preprocessing stage for other minilanguages.

Unix makefiles, which are designed to automate build processes, express dependency relationships between source and derived files[79] and the commands required to make each derived file from its sources. When you run make, it uses those declarations to walk the implied tree of dependencies, doing the least work necessary to bring your build up to date. Like *yacc* and *lex* specifications, makefiles are a declarative minilanguage; they set up constraints that imply actions performed on an interpretive context (in this case, the portion of the file system where the source and generated files live). We'll return to makefiles in Chapter 15.

XSLT, the language used to describe transformations of XML, is at the high end of complexity for declarative minilanguages. It's complex enough that it's not normally thought of as a minilanguage at all, but it shares some important characteristic of such languages which we'll examine when we look at it in more detail below.

The spectrum of minilanguages ranges from declarative (with implicit actions) to imperative (with explicit actions). The run-control syntax of fetchmail(1) can be viewed as either a very weak imperative language or a declarative language with implied control flow. The troff and PostScript

[79] For less technical readers: the compiled form of a C program is derived from its C source form by compilation and linkage. The PostScript version of a troff document is derived from the troff source; the command to make the former from the latter is a *troff* invocation. There are many other kinds of derivation; makefiles can express almost all of them.

typesetting languages are imperative languages with a lot of special-purpose domain expertise baked into them.

Some task-specific imperative minilanguages start to border on being general-purpose interpreters. They reach this level when they are explicitly *Turing-complete*—that is, they can do both conditionals and loops (or recursion)[80] with features that are designed to be used as control structures. Some languages, by contrast, are only accidentally Turing-complete — they have features that can be used to implement control structures as a sort of side effect of what they are actually designed to do.

The bc(1) and dc(1) interpreters we looked at in Chapter 7 are good examples of specialized imperative minilanguages that are explicitly Turing-complete.

We are over the border into general-purpose interpreters when we reach languages like Emacs Lisp and JavaScript that are designed to be full programming languages run in specialized contexts. We'll have more to say about these when we discuss embedded scripting languages later on.

The spectrum in interpreters is one of increasing generality; the flip side of this is that a more general-purpose interpreter embodies fewer assumptions about the context in which it runs. With increasing generality there usually comes a richer ontology of data types. Shell and Tcl have relatively simple ontologies; Perl, Python, and Java more complex ones. We'll return to these general-purpose languages in Chapter 14.

Applying Minilanguages

Designing with minilanguages involves two distinct challenges. One is having existing minilanguages handy in your toolkit, and recognizing when they can be applied as-is. The other is knowing when it is appropriate to design a custom minilanguage for an application. To help you develop both aspects of your design sense, about half of this chapter will consist of case studies.

Case Study: *sng*

In Chapter 6 we looked at sng(1), which translates between PNG and an editable all-text representation of the same bits. The SNG data-file format is worth reexamining for contrast here because

[80] Any Turing-complete language could theoretically be used for general-purpose programming, and is theoretically exactly as powerful as any other Turing-complete language. In practice, some Turing-complete languages would be far too painful to use for anything outside a specified and narrow problem domain.

it is not quite a domain-specific minilanguage. It describes a data layout, but doesn't associate any implied sequence of actions with the data.

SNG does, however, share one important characteristic with domain-specific minilanguages that binary structured data formats like PNG do not — transparency. Structured data files make it possible for editing, conversion, and generation tools to cooperate without knowing about each others' design assumptions other than through the medium of the minilanguage. What SNG adds is that, like a domain-specific minilanguage, it's designed to be easy to parse by eyeball and edit with general-purpose tools.

Case Study: Regular Expressions

A kind of specification that turns up repeatedly in Unix tools and scripting languages is the *regular expression* ('regexp' for short). We consider it here as a declarative minilanguage for describing text patterns; it is often embedded in other minilanguages. Regexps are so ubiquitous that the are hardly thought of as a minilanguage, but they replace what would otherwise be huge volumes of code implementing different (and incompatible) search capabilities.

This introduction skates over some details like POSIX extensions, back-references, and internationalization features; for a more complete treatment, see *Mastering Regular Expressions* [Friedl].

Regular expressions describe patterns that may either match or fail to match against strings. The simplest regular-expression tool is grep(1), a filter that passes through to its output every line in its input matching a specified regexp. Regexp notation is summarized in Table 8.1.

Table 8.1. Regular-expression examples.

Regexp	Matches
`"x.y"`	`x` followed by any character followed by `y`.
`"x\.y"`	`x` followed by a literal period followed by `y`.
`"xz?y"`	`x` followed by at most one `z` followed by `y`; thus, `"xy"` or `"xzy"` but not `"xz"` or `"xdy"`.
`"xz*y"`	`x` followed by any number of instances of `z`, followed by `y`; thus, `"xy"` or `"xzy"` or `"xzzzy"` but not `"xz"` or `"xdy"`.
`"xz+y"`	`x` followed by one or more instances of `z`, followed by `y`; thus, `"xzy"` or `"xzzy"` but not `"xy"` or `"xz"` or `"xdy"`.
`"s[xyz]t"`	`s` followed by any of the characters `x` or `y` or `z`, followed by `t`; thus, `"sxt"` or `"syt"` or `"szt"` but not `"st"` or `"sat"`.
`"a[x0-9]b"`	`a` followed by either `x` or characters in the range `0`–`9`, followed by `b`; thus, `"axb"` or `"a0b"` or `"a4b"` but not `"ab"` or `"aab"`.
`"s[^xyz]t"`	`s` followed by any character that is not `x` or `y` or `z`,

There are a number of minor variants of regexp notation:

1. *Glob expressions.* This is the limited set of wildcard conventions used by early Unix shells for filename matching. There are only three wildcards: `*`, which matches any sequence of characters (like `.*` in the other variants); `?`, which matches any single character (like `.` in the other variants); and `[...]`, which matches a character class just as in the other variants. Some shells (*csh*, *bash*, *zsh*) later added `{}` for alternation. Thus, `x{a,b}c` matches `xac` or `xbc` but not `xc`. Some shells further extend globs in the direction of extended regular expressions.

2. *Basic regular expressions.* This is the notation accepted by the original grep(1) utility for extracting lines matching a given regexp from a file. The line editor ed(1), the stream editor sed(1), also use these. Old Unix hands think of these as the basic or 'vanilla' flavor of regexp; people first exposed to the more modern tools tend to assume the extended form described next.

3. *Extended regular expressions.* This is the notation accepted by the extended grep utility egrep(1) for extracting lines matching a given regexp from a file. Regular expressions in Lex and the *Emacs* editor are very close to the *egrep* flavor.

4. *Perl regular expressions.* This is the notation accepted by Perl and Python regexp functions. These are quite a bit more powerful than the *egrep* flavor.

Now that we've looked at some motivating examples, Table 8.2 is a summary of the standard regular-expression wildcards. Note: we're not including the glob variant in this table, so a value of "All" implies only all three of the basic, extended/Emacs, and Perl/Python variants.[81]

[81]The POSIX standard for regular expressions introduces some symbolic ranges like `[[:lower;;]]` and `[[:digit:]]`, and some specific tools have extra wildcards not covered here, but these will suffice to interpret most regexps.

Table 8.2. Introduction to regular-expression operations.

Wildcard	Supported in	Matches
`\`	All	Escape next character. Toggles whether following punctuation is treated as a wildcard or not. Following letters or digits are interpreted in various different ways depending on the program.
`.`	All	Any character.
`^`	All	Beginning of line
`$`	All	End of line
`[...]`	All	Any of the characters between the brackets
`[^...]`	All	Any characters *except those* between the brackets.
`*`	All	Accept any number of instances of the previous element.
`?`	egrep/Emacs, Perl/Python	Accept zero or one instances of the previous element.
`+`	egrep/Emacs, Perl/Python	Accept one or more instances of the previous element.
`{n}`	egrep, Perl/Python; as `\{n\}` in Emacs	Accept exactly `n` repetitions of the previous element. Not supported by some older regexp engines.
`{n,}`	egrep, Perl/Python; as `\{n,\}` in Emacs	Accept `n` or more repetitions of the previous element. Not supported

Design practice in new languages with regexp support has stabilized on the Perl/Python variant. It is more transparent than the others, notably because backlash before a non-alphanumeric character always means that character as a literal, so there is much less confusion about how to quote elements of regexps.

Regular expressions are an extreme example of how concise a minilanguage can be. Simple regular expressions express recognition behavior that would otherwise have to be implenented with hundreds of lines of fussy, bug-prone code.

Case Study: *Glade*

Glade is an interface builder for the open-source GTK toolkit library for X.[82] *Glade* allows you to develop a GUI interface by interactively picking, placing, and modifying widgets on an interface panel. The GUI editor produces an XML file describing the interface; this, in turn, can be fed to one of several code generators that will actually grind out C, C++, Python or Perl code for the interface. The generated code then calls functions you write to supply behavior to the interface.

Glade's XML format for describing GUIs is a good example of a simple domain-specific minilanguage. See Example 8.1 for a “Hello, world!” GUI in Glade format.

Example 8.1. Glade “Hello, World”.

```
<?xml version="1.0"?>
<GTK-Interface>

<widget>
  <class>GtkWindow</class>
  <name>HelloWindow</name>
  <border_width>5</border_width>
  <Signal>
    <name>destroy</name>
    <handler>gtk_main_quit</handler>
  </Signal>
```

[82]For non-Unix programmers, an X toolkit is a graphics library that supplies GUI widgets (like labels, buttons, and pull-down menus) to the programs that link to it. Under most other graphical operating systems, the OS supplies one toolkit that everyone uses. Unix and X support multiple toolkits; this is part of the separation of policy from mechanism that we called out as a design goal of X in Chapter 1. GTK and Qt are the two most popular open-source X toolkits.

```
  <title>Hello</title>
  <type>GTK_WINDOW_TOPLEVEL</type>
  <position>GTK_WIN_POS_NONE</position>
  <allow_shrink>True</allow_shrink>
  <allow_grow>True</allow_grow>
  <auto_shrink>False</auto_shrink>

  <widget>
   <class>GtkButton</class>
   <name>Hello World</name>
   <can_focus>True</can_focus>
   <Signal>
    <name>clicked</name>
    <handler>gtk_widget_destroy</handler>
    <object>HelloWindow</object>
   </Signal>
   <label>Hello World</label>
  </widget>
</widget>

</GTK-Interface>
```

A valid specification in *Glade* format implies a repertoire of actions by the GUI in response to user behavior. The *Glade* GUI treats these specifications as structured data files; *Glade* code generators, on the other hand, use them to write programs implementing a GUI. For some languages (including Python), there are runtime libraries that allow you to skip the code-generation step and simply instantiate the GUI directly at runtime from the XML specification (interpreting Glade markup, rather than compiling it to the host language). Thus, you get the choice of trading space efficiency for startup speed or vice versa.

Once you get past the verbosity of XML, *Glade* markup is a fairly simple language. It does just two things: declare GUI-widget hierarchies and associate properties with widgets. You don't actually have to know a lot about how *glade* works to read the specification above. In fact, if you have any experience programming in GUI toolkits, reading it will immediately give you a fairly good visualization of what *glade* does with the specification. (Hands up everyone who predicted that this particular specification will give you a single button widget in a window frame.)

This kind of transparency and simplicity is the mark of a good minilanguage design. The mapping between the notation and domain objects is very clear. The relationships between objects are expressed directly, rather than through name references or some other sort of indirection that you have to think to follow.

The ultimate functional test of a minilanguage like this one is simple: can I hack it without reading the manual? For a significant range of cases, the *Glade* answer is yes. For example, if you know the C-level constants that GTK uses to describe window-positioning hints, you'll recognize `GTK_WIN_POS_NONE` as one and instantly be able to change the positioning hint associated with this GUI.

The advantage of using *Glade* should be clear. It specializes in code generation so you don't have to. That's one less routine task you have to hand-code, and one fewer source of hand-coded bugs.

More information, including source code and documentation and links to sample applications, is available at the Glade project page [http://glade.gnome.org/]. *Glade* has been ported to Windows.

Case Study: *m4*

The m4(1) macro processor interprets a declarative minilanguage for describing transformations of text. An *m4* program is a set of macros that specifies ways to expand text strings into other strings. Applying those declarations to an input text with *m4* performs macro expansion and yields an output text. (The C preprocessor performs similar services for C compilers, though in a rather different style.)

Example 8.2 shows an *m4* macro that directs *m4* to expand each occurrence of the string "OS" in its input into the string "operating system" on output. This is a trivial example; *m4* supports macros with arguments that can be used to do more than transform one fixed string into another. Typing **info m4** at your shell prompt will probably display on-line documentation for this language.

Example 8.2. A sample *m4* macro.

```
define(‘OS’, ‘operating system’)
```

The *m4* macro language supports conditionals and recursion. The combination can be used to implement loops, and this was intended; *m4* is deliberately Turing-complete. But actually trying to use *m4* as a general-purpose language would be deeply perverse.

The *m4* macro processor is usually employed as a preprocessor for minilanguages that lack a built-in notion of named procedures or a built-in file-inclusion feature. It's an easy way to extend the syntax of the base language so the combination with *m4* supports both these features.

One well-known use of *m4* has been to clean up (or at least effectively hide) another minilanguage design that was called out as a bad example earlier in this chapter. Most system administrators now generate their `sendmail.cf` configuration files using an *m4* macro package supplied with the *sendmail* distribution. The macros start from feature names (or name/value pairs) and generate the corresponding (much uglier) strings in the *sendmail* configuration language.

Use *m4* with caution, however. Unix experience has taught minilanguage designers to be wary of macro expansion,[83] for reasons we'll discuss later in the chapter.

Case Study: XSLT

XSLT, like *m4* macros, is a language for describing transformations of a text stream. But it does much more than simple macro substitution; it describes transformations of XML data, including query and report generation. It is the language used to write XML stylesheets. For practical applications, see the description of XML document processing in Chapter 18. XSLT is described by a World Wide Web Consortium standard and has several open-source implementations.

XSLT and *m4* macros are both purely declarative and Turing-complete, but XSLT supports only recursions and not loops. It is quite complex, certainly the most difficult language to master of any in this chapter's case studies — and probably the most difficult of any language mentioned in this book.[84]

Despite its complexity, XSLT really is a minilanguage. It shares important (though not universal) characteristics of the breed:

- A restricted ontology of types, with (in particular) no analog of record structures or arrays.
- Restricted interface to the rest of the world. XSLT processors are designed to filter standard input to standard output, with a limited ability to read and write files. They can't open sockets or run subcommands.

[83] Whether or not "macro expansion" should be spelled "macroexpansion" is a matter for some dispute. The latter is found mainly among Lisp programmers.

[84] It is not clear that XSLT could be any simpler and still do its job, however, so we cannot characterize it as a bad design.

Example 8.3. A sample XSLT program.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
            version="1.0">
 <xsl:output method="xml"/>
 <xsl:template match="*">
   <xsl:element name="{name()}">
     <xsl:for-each select="@*">
       <xsl:element name="{name()}">
         <xsl:value-of select="."/>
       </xsl:element>
     </xsl:for-each>
     <xsl:apply-templates select="*|text()"/>
   </xsl:element>
 </xsl:template>
</xsl:stylesheet>
```

The program in Example 8.3 transforms an XML document so that each attribute of every element is transformed into a new tag pair directly enclosed by that element, with the attribute value as the tag pair's content.

We've included a glance at XSLT here partly to illustrate the point that 'declarative' does not imply either 'simple' or 'weak', and mostly because if you have to work with XML documents, you will someday have to face the challenge that is XSLT.

XSLT: Mastering XML Transformations [Tidwell] is a good introduction to the language. A brief tutorial with examples is available on the Web.[85]

Case Study: The Documenter's Workbench Tools

The troff(1) typesetting formatter was, as we noted in Chapter 2, Unix's original killer application. *troff* is the center of a suite of formatting tools (collectively called Documenter's Workbench or

[85] XSL Concepts and Practical Use [http://nwalsh.com/docs/tutorials/xsl/xsl/slides.html].

DWB), all of which are domain-specific minilanguages of various kinds. Most are either preprocessors or postprocessors for troff markup. Open-source Unixes host an enhanced implementation of Documenter's Workbench called groff(1), from the Free Software Foundation.

We'll examine *troff* in more detail in Chapter 18; for now, it's sufficient to note that it is a good example of an imperative minilanguage that borders on being a full-fledged interpreter (it has conditionals and recursion but not loops; it is accidentally Turing-complete).

The postprocessors ('drivers' in DWB terminology) are normally not visible to *troff* users. The original troff emitted codes for the particular typesetter the Unix development group had available in 1970; later in the 1970s these were cleaned up into a device-independent minilanguage for placing text and simple graphics on a page. The postprocessors translate this language (called "ditroff" for "device-independent troff") into something modern imaging printers can actually accept — the most important of these (and the modern default) is PostScript.

The preprocessors are more interesting, because they actually add capabilities to the troff language. There are three common ones: tbl(1) for making tables, eqn(1) for typesetting mathematical equations, and pic(1) for drawing diagrams. Less used, but still live, are grn(1) for graphics, and refer(1) and bib(1) for formatting bibliographies. Open-source equivalents of all of these ship with *groff*. The grap(1) preprocessor provided a rather versatile plotting facility; there is an open-source implementation separate from *groff*.

Some other preprocessors have no open-source implementation and are no longer in common use. Best known of these was ideal(1), for graphics. A younger sibling of the family, chem(1), draws chemical structural formulas; it is available as part of Bell Labs's netlib code.[86]

Each of these preprocessors is a little program that accepts a minilanguage and compiles it into troff requests. Each one recognizes the markup it is supposed to interpret by looking for a unique start and end request, and passes through unaltered any markup outside those (*tbl* looks for .TS/.TE, *pic* looks for .PS/.PE, etc.). Thus, most of the preprocessors can normally be run in any order without stepping on each other. There are some exceptions: in particular, *chem* and *grap* both issue *pic* commands, and so must come before it in the pipeline.

```
cat thesis.ms | chem | tbl | refer | grap | pic | eqn \
                                  | groff -Tps >thesis.ps
```

[86] http://www.netlib.org/

The preceding is a full-Monty example of a Documenter's Workbench processing pipeline, for a hypothetical thesis incorporating chemical formulas, mathematical equations, tables, bibliographies, plots, and diagrams. (The cat(1) command simply copies its input or a file argument to its output; we use it here to emphasize the order of operations.) In practice modern troff implementations tend to support command-line options that can invoke at least tbl(1), eqn(1) and pic(1), so it isn't necessary to write such an elaborate pipeline. Even if it were, these sorts of build recipes are normally composed just once and stashed away in a makefile or shellscript wrapper for repeated use.

The document markup of Documenter's Workbench is in some ways obsolete, but the range of problems these preprocessors address gives some indication of the power of the minilanguage model — it would be extremely difficult to embed equivalent knowledge into a WYSIWYG word processor. There are some ways in which modern XML-based document markups and toolchains are still, in 2003, playing catch-up with capabilities that Documenter's Workbench had in 1979. We'll discuss these issues in more detail in Chapter 18.

The design themes that gave Documenter's Workbench so much power should by now be familiar ones; all the tools share a common text-stream representation of documents, and the formatting system is broken up into independent components that can be debugged and improved separately. The pipeline architecture supports plugging in new, experimental preprocessors and postprocessors without disturbing old ones. It is modular and extensible.

The architecture of Documenter's Workbench as a whole teaches us some things about how to fit multiple specialist minilanguages into a cooperating system. One preprocessor can build on another. Indeed, the Documenter's Workbench tools were an early exemplar of the power of pipes, filtering, and minilanguages that influenced a lot of later Unix design by example. The design of the individual preprocessors has more lessons to teach about what effective minilanguage designs look like.

One of these lessons is negative. Sometimes users writing descriptions in the minilanguages do unclean things with low-level troff markup inserted by hand. This can produce interactions and bugs that are hard to diagnose, because the generated troff coming out of the pipeline is not visible — and would not be readable if it were. This is analogous to the sorts of bugs that happen in code that mixes C with snippets of in-line assembler. It might have been better to separate the language layers more completely, if that were possible. Minilanguage designers should take note of this.

All the preprocessor languages (though not troff markup itself) have relatively clean, shell-like syntaxes that follow many of the conventions we described in Chapter 5 for the design of data-file formats. There are a few embarrassing exceptions; notably, tbl(1) defaults to using a tab as a field separator between table columns, replicating an infamous botch in the design of make(1) and causing annoying bugs when editors or other tools invisibly change the composition of whitespace.

While troff itself is a specialized imperative language, one theme that runs through at least three of the Documenter's Workbench minilanguages is declarative semantics: doing layout from constraints. This is an idea that shows up in modern GUI toolkits as well — that, instead of giving pixel coordinates for graphical objects, what you really want to do is declare spatial relationships among them ("widget A is above widget B, which is to the left of widget C") and have your software compute a best-fit layout for A, B, and C according to those constraints.

The pic(1) program uses this approach to lay out elements for diagrams. The language taxonomy diagram at Figure 8.1 was produced with the *pic* source code in Example 8.4[87] run through *pic2graph*, one of our case studies in Chapter 7.

Example 8.4. Taxonomy of languages — the *pic* source.

```
# Minilanguage taxonomy
#
# Base ellipses
define smallellipse {ellipse width 3.0 height 1.5}
M: ellipse width 3.0 height 1.8 fill 0.2
line from M.n to M.s dashed
D: smallellipse() with .e at M.w + (0.8, 0)
line from D.n to D.s dashed
I: smallellipse() with .w at M.e - (0.8, 0)
#
# Captions
"" "Data formats" at D.s
"" "Minilanguages" at M.s
"" "Interpreters" at I.s
#
# Heads
arrow from D.w + (0.4, 0.8) to D.e + (-0.4, 0.8)
"flat to structured" "" at last arrow.c
```

[87] It is also quite traditional for Unix books that describe pic(1) to include their own illustrations as coding examples.

```
arrow from M.w + (0.4, 1.0) to M.e + (-0.4, 1.0)
"declarative to imperative" "" at last arrow.c
arrow from I.w + (0.4, 0.8) to I.e + (-0.4, 0.8)
"less to more general" "" at last arrow.c
#
# The arrow of loopiness
arrow from D.w + (0, 1.2) to I.e + (0, 1.2)
"increasing loopiness" "" at last arrow.c
#
# Flat data files
"/etc/passwd" ".newsrc" at 0.5 between D.c and D.w
# Structured data files
"SNG" at 0.5 between D.c and M.w
# Datafile/minilanguage borderline cases
"regexps" "Glade" at 0.5 between M.w and D.e
# Declarative minilanguages
"m4" "Yacc" "Lex" "make" "XSLT" "pic" "tbl" "eqn" \
 at 0.5 between M.c and D.e
# Imperative minilanguages
"fetchmail" "awk" "troff" "Postscript" at 0.5 between M.c and I.w
# Minilanguage/interpreter borderline cases
"dc" "bc" at 0.5 between I.w and M.e
# Interpreters
"Emacs Lisp" "JavaScript" at 0.25 between M.e and I.e
"sh" "tcl" at 0.55 between M.e and I.e
"Perl" "Python" "Java" at 0.8 between M.e and I.e
```

This is a very typical Unix minilanguage design, and as such has some points of interest even on the purely syntactic level. Notice how much it looks like a shell program: # leads comments, and the syntax is obviously token-oriented with the simplest possible convention for strings. The designer of pic(1) knew that Unix programmers expect minilanguage syntaxes to look like this unless there is a strong and specific reason they should not. The Rule of Least Surprise is in full operation here.

It probably doesn't take a lot of effort to discern that the first line of code is a macro definition; the later references to `smallellipse()` encapsulate a repeated design element of the diagram. Nor will it take much scrutiny to deduce that `box invis` declares a box with invisible borders, actually just a frame for text to be stacked inside. The `arrow` command is equally obvious.

With these as clues and one eye on the actual diagram, the meaning of the remaining pieces of the syntax (position references like `M.s` and constructions like `last arrow` or `at 0.25 between M.e and I.e` or the addition of vector offsets to a location) should become rapidly apparent. As with Glade markup and *m4*, an example like this one can teach a good bit of the language without any reference to a manual (a compactness property troff(1) markup, unfortunately, does *not* have).

The example of pic(1) reflects a common design theme in minilanguages, which we also saw reflected in Glade — the use of a minilanguage interpreter to encapsulate some form of constraint-based reasoning and turn it into actions. We could actually choose to view pic(1) as an imperative language rather than a declarative one; it has elements of both, and the dispute would quickly grow theological.

The combination of macros with constraint-based layout gives pic(1) the ability to express the structure of diagrams in a way that more modern vector-based markups like SVG cannot. It is therefore fortunate that one effect of the Documenter's Workbench design is to make it relatively easy to keep pic(1) useful outside the DWB context. The *pic2graph* script we used as a case study in Chapter 7 was an ad-hoc way to accomplish this, using the retrofitted PostScript capability of groff(1) as a half-way step to a modern bitmap format.

A cleaner solution is the pic2plot(1) utility distributed with the GNU plotutils package, which exploited the internal modularity of the GNU pic(1) code. The code was split into a parsing front end and a back end that generated troff markup, the two communicating through a layer of drawing primitives. Because this design obeyed the Rule of Modularity, *pic2plot(1)* implementers were able to split off the GNU *pic* parsing stage and reimplement the drawing primitives using a modern plotting library. Their solution has the disadvantage, however, that text in the output is generated with fonts built into *pic2plot* that won't match those of troff.

Case Study: *fetchmail* Run-Control Syntax

See Example 8.5 for an example.

Example 8.5. Synthetic example of a `fetchmailrc`.

```
# Poll this site first each cycle.
poll pop.provider.net proto pop3
   user "jsmith" with pass "secret1" is "smith" here
   user jones with pass "secret2" is "jjones" here with options keep
```

```
# Poll this site second, unless Lord Voldemort zaps us first.
poll billywig.hogwarts.com with proto imap:
   user harry_potter with pass "floo" is harry_potter here

# Poll this site third in the cycle.
# Password will be fetched from ~/.netrc
poll mailhost.net with proto imap:
   user esr is esr here
```

This run-control file can be viewed as an imperative minilanguage. There is an implied flow of execution: cycle through the list of poll commands repeatedly (sleeping for a while at the end of each cycle), and for each site entry collect mail for each associated user in sequence. It is far from being general-purpose; all it can do is sequence the program's polling behavior.

As with pic(1), one could choose to view this minilanguage as either declarations or a very weak imperative language, and argue endlessly over the distinction. On the one hand, it has neither conditionals nor recursion nor loops; in fact, it has no explicit control structures at all. On the other hand, it does describe actions rather than just relationships, which distinguishes it from a purely declarative syntax like Glade GUI descriptions.

Run-control minilanguages for complex programs often straddle this border. We're making a point of this fact because not having explicit control structures in an imperative minilanguage can be a tremendous simplification if the problem domain lets you get away with it.

One notable feature of `.fetchmailrc` syntax is the use of optional noise keywords that are supported simply in order to make the specifications read a bit more like English. The 'with' keywords and single occurrence of 'options' in the example aren't actually necessary, but they help make the declarations easier to read at a glance.

The traditional term for this sort of thing is *syntactic sugar*; the maxim that goes with this is a famous quip that "syntactic sugar causes cancer of the semicolon".[88] Indeed, syntactic sugar needs to be used sparingly lest it obscure more than help.

In Chapter 9 we'll see how data-driven programming helps provide an elegant solution to the problem of editing *fetchmail* run-control files through a GUI.

[88]The line is owed to Alan Perlis, who did some of the pioneering work in software modularity around 1970. The semicolon in question was the statement separator or terminator in various Algol-descended languages, including Pascal and C.

Case Study: *awk*

The *awk* minilanguage is an old-school Unix tool, formerly much used in shellscripts. Like *m4*, it's intended for writing small but expressive programs to transform textual input into textual output. Versions ship with all Unixes, several in open source; the command **info gawk** at your Unix shell prompt is quite likely to take you to on-line documentation.

Programs in *awk* consist of pattern/action pairs. Each pattern is a *regular expression*, a concept we'll describe in detail in Chapter 9. When an *awk* program is run, it steps through each line of the input file. Each line is checked against every pattern/action pair in order. If the pattern matches the line, the associated action is performed.

Each action is coded in a language resembling a subset of C, with variables and conditionals and loops and an ontology of types including integers, strings, and (unlike C) dictionaries.[89]

The *awk* action language is Turing-complete, and can read and write files. In some versions it can open and use network sockets. But *awk* has primarily seen use as a report generator, especially for interpreting and reducing tabular data. It is seldom used standalone, but rather embedded in scripts. There is an example *awk* program in the case study on HTML generation included in Chapter 9.

A case study of *awk* is included to point out that it is *not* a model for emulation; in fact, since 1990 it has largely fallen out of use. It has been superseded by new-school scripting languages—notably Perl, which was explicitly designed to be an *awk* killer. The reasons are worthy of examination, because they constitute a bit of a cautionary tale for minilanguage designers.

The *awk* language was originally designed to be a small, expressive special-purpose language for report generation. Unfortunately, it turns out to have been designed at a bad spot on the complexity-vs.-power curve. The action language is noncompact, but the pattern-driven framework it sits inside keeps it from being generally applicable — that's the worst of both worlds. And the new-school scripting languages can do anything *awk* can; their equivalent programs are usually just as readable, if not more so.

> Awk has also fallen out of use because more modern shells have floating point arithmetic, associative arrays, RE pattern matching, and substring capabilities, so

[89] For those who have never programmed in a modern scripting language, a dictionary is a lookup table of key-to-value associations, often implemented through a hash table. C programmers spend a lot of their coding time implementing dictionaries in various elaborate ways.

that equivalents of small awk scripts can be done without the overhead of process creation.

—

<author>DavidKorn</author>

For a few years after the release of Perl in 1987, *awk* remained competitive simply because it had a smaller, faster implementation. But as the cost of compute cycles and memory dropped, the economic reasons for favoring a special-purpose language that was relatively thrifty with both lost their force. Programmers increasingly chose to do awklike things with Perl or (later) Python, rather than keep two different scripting languages in their heads.[90] By the year 2000 *awk* had become little more than a memory for most old-school Unix hackers, and not a particularly nostalgic one.

Falling costs have changed the tradeoffs in minilanguage design. Restricting your design's capabilities to buy compactness may still be a good idea, but doing so to economize on machine resources is a bad one. Machine resources get cheaper over time, but space in programmers' heads only gets more expensive. Modern minilanguages can either be general but noncompact, or specialized but very compact; specialized but noncompact simply won't compete.

Case Study: PostScript

PostScript is a minilanguage specialized for describing typeset text and graphics to imaging devices. It is an import into Unix, based on design work done at the legendary Xerox Palo Alto Research Center along with the earliest laser printers. For years after its first commercial release in 1984, it was available only as a proprietary product from Adobe, Inc., and was primarily associated with Apple computers. It was cloned under license terms very close to open-source in 1988, and has since become the de-facto standard for printer control under Unix. A fully open-source version is shipped with most most modern Unixes.[91] A good technical introduction to PostScript is also available.[92]

PostScript bears some functional resemblance to troff markup; both are intended to control printers and other imaging devices, and both are normally generated by programs or macro packages rather than being hand-written by humans. But where troff requests are a jumped-up set of format-control codes with some language features tacked on as an afterthought, PostScript was designed from the ground up as a language and is far more expressive and powerful. The main thing that makes

[90] I was at one time an *awk* wizard, but I had to be reminded by someone else that the language was applicable to the HTML-generation problem where this book's only *awk* example occurs.
[91] There is a GhostScript Project site [http://www.cs.wisc.edu/~ghost/].
[92] A First Guide To PostScript [http://www.cs.indiana.edu/docproject/programming/postscript/postscript.html].

Postscript useful is that algorithmic descriptions of images written in it are far smaller than the bitmaps they render to, and so take up less storage and communication bandwidth.

PostScript is explicitly Turing-complete, supporting conditionals and loops and recursion and named procedures. The ontology of types includes integers, reals, strings, and arrays (each element of an array may be of any type) but no equivalent of structures. Technically, PostScript is a stack-based language; arguments of PostScript's primitive procedures (operators) are normally popped off a push-down stack of arguments, and the result(s) are pushed back onto it.

There are about 40 basic operators out of a total of around 400. The one that does most of the work is **show**, which draws a string onto the page. Others set the current font, change the gray level or color, draw lines or arcs or Bezier curves, fill closed regions, set clipping regions, etc. A PostScript interpreter is supposed to be able to interpret these commands into bitmaps to be thrown on a display or print medium.

Other PostScript operators implement arithmetic, control structures, and procedures. These allow repetitive or stereotyped images (such as text, which is composed of repeated letterforms) to be expressed as programs that combine images. Part of the utility of PostScript comes from the fact that PostScript programs to print text or simple vector graphics are much less bulky than the bitmaps the text or vectors render to, are device-resolution independent, and travel more quickly over a network cable or serial line.

Historically, PostScript's stack-based interpretation resembles a language called FORTH, originally designed to control telescope motors in real time, which was briefly popular in the 1980s. Stack-based languages are famous for supporting extremely tight, economical coding and infamous for being difficult to read. PostScript shares both traits.

PostScript is often implemented as firmware built into a printer. The open-source implementation Ghostscript can translate PostScript to various graphics formats and (weaker) printer-control languages. Most other software treats PostScript as a final output format, meant to be handed to a PostScript-capable imaging device but not edited or eyeballed.

PostScript (either in the original or the trivial variant EPSF, with a bounding box declared around it so it can be embedded in other graphics) is a very well designed example of a special-purpose control language and deserves careful study as a model. It is a component of other standards such as PDF, the Portable Document Format.

Case Study: *bc* and *dc*

We first examined bc(1) and dc(1) in Chapter 7 as a case study in shellouts. They are examples of domain-specific minilanguages of the imperative type.

> *dc* is the oldest language on Unix; it was written on the PDP-7 and ported to the PDP-11 before Unix [itself] was ported.
>
> —
> <author>KenThompson</author>

The domain of these two languages is unlimited-precision arithmetic. Other programs can use them to do such calculations without having to worry about the special techniques needed to do those calculations.

> In fact, the original motivation for dc had nothing to do with providing a general-purpose interactive calculator, which could have been done with a simple floating-point program. The motivation was Bell Labs' long interest in numerical analysis: calculating constants for numerical algorithms, *accurately* is greatly aided by being able to work to much higher precision than the algorithm itself will use. Hence dc's arbitrary-precision arithmetic.
>
> —
> <author>HenrySpencer</author>

Like SNG and *Glade* markup, one of the strengths of both of these languages is their simplicity. Once you know that dc(1) is a reverse-Polish-notation calculator and bc(1) an algebraic-notation calculator, very little about interactive use of either of these languages is going to be novel. We'll return to the importance of the Rule of Least Surprise in interfaces in Chapter 11.

These minilanguages have both conditionals and loops; they are Turing-complete, but have a very restricted ontology of types including only unlimited-precision integers and strings. This puts them in the borderland between interpreted minilanguages and full scripting languages. The programming features have been designed not to intrude on the common use as a calculator; indeed, most *dc*/*bc* users are probably unaware of them.

Normally, *dc*/*bc* are used conversationally, but their capacity to support libraries of user-defined procedures gives them an additional kind of utility — programmability. This is actually the most important advantage of imperative minilanguages, one that we observed in the case study of the Documenter's Workbench tools to be very powerful whether or not a program's normal mode is conversational; you can use them to write high-level programs that embody task-specific intelligence.

Because the interface of *dc*/*bc* is so simple (send a line containing an expression, get back a line containing a value) other programs and scripts can easily get access to all these capabilities by calling these programs as slave processes. Example 8.6 is one famous example, an implementation of the Rivest-Shamir-Adelman public-key cipher in Perl that was widely published in signature blocks and on T-shirts as a protest against U.S. export retrictions on cryptography, c. 1995; it shells out to *dc* to do the unlimited-precision arithmetic required.

Example 8.6. RSA implementation using *dc*.

```
print pack"C*",split/\D+/,‘echo "16iII*o\U@{$/=$z;[(pop,pop,unpack
"H*",<>)]}\EsMsKsN0[lN*1lK[d2%Sa2/d0<X+d*lMLa^*lN%0]dsXx++\
lMlN/dsM0<J]dsJxp"|dc‘
```

Case Study: Emacs Lisp

Rather than merely being run as a slave process to accomplish specific tasks, a special-purpose interpreted language can become the core of an entire architecture; we'll consider the advantages and disadvantages of this approach in Chapter 13. troff requests were an early example; today, the *Emacs* editor is one of the best-known and most powerful modern ones. It's built around a dialect of Lisp with primitives for both describing actions on editing buffers and controlling slave processes.

The fact that Emacs is built around a powerful language for describing editing actions or front ends for other programs means that it can be used for many other things besides ordinary editing. We'll examine the applications of Emacs's task-specific intelligence for day-to-day program development (compilation, debugging, version control) in Chapter 15. Emacs 'modes' are user-defined libraries — programs written in Emacs Lisp that specialize the editor for a particular job — usually, but not necessarily, one related to editing.

Thus there are specialized modes that know the syntax of a large number of programming languages, and of markup languages like SGML, XML, and HTML. But many people also use Emacs modes to send and receive email (these use Unix system mail utilities as slaves) or Usenet news. Emacs can browse the web, or act as a front-end for various chat programs. There is also a calendaring package, Emacs's own calculator program, and even a fairly wide selection of games written as Emacs

Lisp modes (including a descendant of the famous ELIZA program that simulates a Rogersian psychiatrist).[93]

Case Study: JavaScript

JavaScript is an open-source language designed to be embedded in C programs. Though it is also embedded in web servers, its original and best-known manifestation is client-side JavaScript, which allows you to embed executable code in Web pages to be run by any JavaScript-capable browser. That is the version we will survey here.

JavaScript is a fully Turing-complete interpreted language with integers, real numbers, booleans, strings, and lightweight dictionary-based objects resembling those of Python. Values are typed, but variables can hold any type; conversions between types are automatic in many contexts. Syntactically JavaScript resembles Java with some influence from Perl, and features Perl-like regular expressions.

Despite all these features, client-side JavaScript is not quite a general-purpose language. Its capabilities are severely restricted to prevent attacks on the browser user through Web pages containing JavaScript code. It can accept input from the user and generate or modify Web pages, but it cannot directly alter the contents of disk files and cannot open its own network connections.

Over time, the JavaScript language has become more general and less bound to its client-side environment. This is something that can be expected to happen to any successful specialized language as its possibilities unfold in the minds of developers and users. Client JavaScript now interacts with its environment by reading and writing values in a single special object called the browser DOM (Document Object Model). The language still has some legacy APIs to the browser that don't go through the DOM, but these are deprecated, not present in the ECMA-262 standard for JavaScript, and may not be supported in future versions.

The standard reference for JavaScript is *JavaScript: The Definitive Guide* [FlanaganJavaScript]. Source code is downloadable.[94] JavaScript makes an interesting study for two reasons. First, it's about as close to being a general-purpose language as one can get without actually being there. Second, the binding between client-side JavaScript and its browser environment via a single DOM object is well designed, and could serve as a model for other embedding situations.

[93]One of the silliest things you can do with a modern Unix machine is run the Eliza mode of Emacs against random quotes from Zippy the Pinhead. **M-x psychoanalyze-pinhead**; type control-G when you've had enough.
[94]Open-source JavaScript implementations in C and Java are available [http://www.mozilla.org/js/].

Designing Minilanguages

When is designing a minilanguage appropriate? We've observed that minilanguages offer a way to push problem specifications to a higher level, and seen how this operates in several case studies. The flip side of this observation is that a minilanguage is likely to be a good approach whenever the domain primitives in your application area are simple and stereotyped, but the ways in which users are likely to want to apply them are fluid and varying.

For some related ideas, find a description of the Alternate Hard And Soft Layers [http://www.c2.com/cgi/wiki?AlternateHardAndSoftLayers] and Scripted Components [http://www.doc.ic.ac.uk/~np2/patterns/scripting/scripting.html] design patterns.

An interesting survey of design styles and techniques in minilanguages is *Notable Design Patterns for Domain-Specific Languages* [Spinellis].

Choosing the Right Complexity Level

The first important thing to bear in mind when designing a minilanguage is, as usual, to keep it as simple as possible. The taxonomy diagram we used to organize the case studies implies a hierarchy of complexity; you want to keep your design as far toward the left-hand edge as possible. If you can get away with designing a structured data file rather than a minilanguage that is going to modify external data when it's interpreted, by all means do so.

One very pragmatic reason to stick with structured data rather than a minilanguage is that in a networked world, embedded minilanguage facilities are subject to abuses that can be inconvenient or even dangerous. JavaScript is a prime example in the 'inconvenient' category; its designers didn't anticipate that it would be used for pop-up advertisements so obnoxious as to create a demand for browser features that suppress JavaScript interpretation.

Microsoft Word macro viruses show how this sort of thing can become actively dangerous, a security hole that costs billions of dollars in downtime and lost productivity annually. It is instructive to note that despite the existence of at least twenty million Unix users worldwide[95] there has never been any Unix equivalent of Windows's frequent macro-virus outbreaks. There are a number of reasons for this, including the fundamentally better security design of Unix; but at least one is the fact that Unix mail agents do *not* default to executing live content in any document that the user views.[96]

[95] 20M is a conservative estimate based on mid-2003 figures from the Linux Counter and elsewhere.
[96] *Kmail*, which we looked at in Chapter 6, won't even chase off-site links in HTML for this reason.

If there is any way that your application's users might end up running programs from untrusted sources, risky features of your application minilanguage might end up having to be suppressed. Languages like Java and JavaScript are explicitly *sandboxed*—that is, they have limited access to their environment not merely to simplify their design but to try to prevent potentially destructive operations by buggy or malicious code.

On the other hand, a lot of bad designs have been botched by designers who failed to face up to the fact that they really needed a minilanguage rather than a data-file format. Too often, language-like features get pasted on as an afterthought. The two most common symptoms of this problem are weak, ad-hoc control structures and poor or nonexistent facilities for declaring procedures.

It's risky to design minilanguages that are only accidentally Turing-complete. If you do this the odds are good that, sometime in the future, some clever fellow is going to think he needs to press your language into doing loops and conditionals for him. Because these are only available in an obfuscated way, he'll produce obfuscated code. The results may be serviceable in the short term, but are likely to be a nightmare for those who come after him.

Minilanguage design is both powerful and esthetically rewarding, but it's also full of similar traps. There are kinds of design in which it is appropriate to take the bottom-up approach of pasting together a bunch of low-level services and worrying about their organization after you have explored the problem domain for a while. One of the virtues of minilanguages is that they can help you get a good design out of bottom-up programming by allowing you to defer some top-down decisions into the control flow of programs in your minilanguage. But if you take a bottom-up approach to the minilanguage design *itself*, you are likely to end up with an ugly syntax reflecting a weak language and a poorly-thought-out implementation.

There are many places in a minilanguage design where small choices make a large difference in the useability and ease of the tool:

> As a language designer, it is a good principle to consider the alternatives to giving an error message. When there is true ambiguity in the intent of the programmer an error message is appropriate, but in many cases the intent is clear, and making the language silently do the right thing is a great boon. A good example is C accommodating an extra comma at the end of an array initializer list, which makes both editing and machine generation of array initializers much easier. Anti-examples are the pickiness of various HTML readers, especially their habit of silently discarding parts of your document because of trivial nesting errors.
>
> —

<author>SteveJohnson</author>

On this issue, as elsewhere, there is no substitute for good taste and engineering judgment. If you're going to design a minilanguage, don't do it halfway. Declarative minilanguages should have a clear, consistent language-like syntax designed to be readable by humans. Imperative ones should add a full range of control structures adapted from language models you can expect your users to be familiar with. Think about the language *as* a language; ask yourself esthetic questions like "Will this be comfortable to program in?" and even "Will it be pleasant to look at?" Here, as elsewhere in software design, David Gelernter's maxim is apt: beauty is the ultimate defense against complexity.

Extending and Embedding Languages

One fundamentally important question is whether you can implement your minilanguage by extending or embedding an existing scripting language. This is often the right way to go for an imperative minilanguage, but much less appropriate for a declarative one.

Sometimes it's possible to write your imperative language simply by coding service functions in an interpreted language, which we'll call the 'host' language for purposes of this discussion. Your minilanguage programs are then just scripts that load your service library and use the host language's control structures and other facilities as a framework. Every facility the host language supplies is one you don't have to write.

This is the easiest way to write a minilanguage. Old-school Lisp programmers (including me) love this technique and use it heavily. It underlies the design of the *Emacs* editor, and has been rediscovered in the new-school scripting languages like Tcl, Python, and Perl. There are drawbacks to it, however.

Your host language may be unable to interface to a code library that you need. Or, internally, its ontology of data types may be inadequate for the kind of computation you need to do. Or, after measuring the performance of a prototype, you discover that it's too slow. When any of these things happen, your solution is usually going to involve coding in C (or C++) and integrating the results into your minilanguage.

The option of extending a scripting language with C code, or of embedding a scripting language in a C program, relies on the existence of scripting languages designed for it. You extend a scripting language by telling it to dynamically load a C library or module in such a way that the C entry points become visible as functions in the extended language. You embed a scripting language in a

C program by sending commands to an instance of the interpreter and receiving the results back as values in C.

Both techniques also rely on the ability to move data between the type ontology of C and the type ontology of your scripting language. Some scripting languages are designed from the ground up to support this. One such is Tcl, which we'll cover in Chapter 14. Another is *Guile*, an open-source dialect of the Lisp variant Scheme. *Guile* is shipped as a library and specifically designed to be embedded in C programs.

It is possible (though in 2003 still rather painful and difficult) to extend or embed Perl. It is very easy to extend Python and only slightly more difficult to embed it; C extension is especially heavily used in the Python world. Java has an interface to call 'native methods' in C, though the practice is explicitly discouraged because it tends to break portability. Most versions of shell are not designed for embeddability and extension, but the Korn shell (ksh93 and later versions) is a notable exception.

There are lots of bad reasons not to piggyback your imperative minilanguage on an existing scripting language. One of the few good ones is that you actually want to implement your own custom grammar for error checking. If that's the case, then see the advice about *yacc* and *lex* below.

Writing a Custom Grammar

For declarative minilanguages, one major question is whether or not you should use XML as a base syntax and specify your grammar as an XML document type. This may well be the right thing for elaborately structured declarative minilanguages, but the same caveats we noted in Chapter 5 about the design of data-file formats apply — XML might be overkill. If you don't use XML, follow the Rule of Least Surprise by supporting the Unix conventions we described for data files (simple token-oriented syntax, supporting C backslash conventions, etc.).

If you do need a custom grammar, *yacc* and *lex* (or their local equivalent in the language you're using) should probably be your best friends, unless the grammar of your language is so simple that hand-coding a recursive-descent parser is trivial. Even then, *yacc* may give you better error recovery, and a *yacc* specification will be easier to modify as the language syntax evolves. See Chapter 9 for a look at the *yacc*- and *lex*-derived tools available in different implementation languages.

Even if you decide you must implement your own syntax, consider what mileage you can get from reusing existing tools. If you need a macro facility, consider whether preprocessing with m4(1) might be the right answer — but consider the cautions in the next section first.

Macros — Beware!

Macro expansion facilities were a favored tactic for language designers in early Unix; the C language has one, of course, and we have seen them show up in some of the more complex special-purpose minilanguages like pic(1). The *m4* preprocessor provides a generic tool for implementing macro-expanding preprocessors.

Macro expansion is easy to specify and implement, and you can do a lot of cute tricks with it. Those early designers appear to have been influenced by experience with assemblers, in which macro facilities were often the only device available for structuring programs.

The strength of macro expansion is that it knows nothing about the underlying syntax of the base language, and can be used to extend that syntax. Unfortunately, this power is very easily abused to produce code that is opaque, surprising, and a fertile source of hard-to-characterize bugs.

In C, the classic example of this sort of problem is a macro such as this:

```
#define max(x, y) x > y ? x : y
```

There are at least two problems with this macro. One is that it can produce surprising results if either of the arguments is an expression including an operator of lower precedence than `>` or `?:`. Consider the expression `max(a = b, ++c)`. If the programmer has forgotten that `max` is a macro, he will be expecting the assignment `a = b` and the preincrement operation on `c` to be executed before the resulting values are passed as arguments to `max`.

But that's not what will happen. Instead, the preprocessor will expand this expression to `a = b > ++c ? a = b : ++c`, which the C compiler's precedence rules make it interpret as `a = (b > ++c ? a = b : ++c)`. The effect will be to assign to `a`!

This sort of bad interaction can be headed off by coding the macro definition more defensively.

```
#define max(x, y) ((x) > (y) ? (x) : (y))
```

With this definition, the expansion would be `((a = b) > (++c) ? (a = b) : (++c))`. This solves one problem — but notice that `c` may be incremented twice! There are subtler versions of this trap, such as passing the macro a function-call with side effects.

In general, interactions between macros and expressions with side effects can lead to unfortunate results that are hard to diagnose. C's macro processor is a deliberately lightweight and simple one; more powerful ones can actually get you in worse trouble.

The *TeX* formatting language (see Chapter 18) well illustrates the general problem. *TeX* is intentionally Turing-complete (it has conditionals, loops, and recursion), but while it can be made to do amazing things, *TeX* code tends to be unreadable and painful to debug. The sources for *LaTeX*, the the most widely used *TeX* macro package, are an instructive example: they're in very good *TeX* style, but even so are extremely difficult to follow.

A minor problem, compared to this one, is that macro expansion tends to screw up error diagnostics. The base language processor generates its error reports relative to the macro expanded text, not the original the programmer is looking at. If the relationship between the two has been obfuscated by macro expansion, the emitted diagnostic can be very difficult to associate with the actual location of the error.

This is especially a problem with preprocessors and macros that can have multiline expansions, conditionally include or exclude text, or otherwise change line numbers in the expanded text.

Macro expansion stages that are built into a language can do their own compensation, fiddling line numbers to refer back to the preexpanded text. The macro facility in pic(1) does this, for example. This problem is more difficult to solve when the macro expansion is done by a preprocessor.

The C preprocessor addresses this problem by emitting `#line` directives whenever it does an inclusion or multiline expansion. The C compiler is expected to interpret these and adjust the line numbers in its error reports accordingly. Unfortunately, *m4* has no such facility.

These are reasons to use macro expansion with extreme caution. One of the long-term lessons of the Unix experience is that macros tend to create more problems than they solve. Modern language and minilanguage designs have moved away from them.

Language or Application Protocol?

Another important question you need to ask is whether your minilanguage engine will be called interactively by other programs, as a slave process. If so, your design should probably look less like a conversational language for human interaction and more like the kind of application protocols we looked at in Chapter 5.

The main difference is how carefully marked the boundaries of transactions are. Human beings are good at spotting where conversational output from a CLI ends, and where the prompt for the next input is. They can use context to tell what's significant and what should be ignored. Computer programs have much more trouble with this. Without either unambiguous end markers on output or advance knowledge of the length of the output, they can't tell when to stop reading.

> Even worse is when a program's input is buffered (often inadvertently, as by stdio). A program that stops overtly reading at the right place can nonetheless eat past it.
>
> —
>
> <author>DougMcIlroy</author>

Programs in which master processes are trying to do interactive things with slaved minilanguages that are not carefully designed around this problem are prone to deadlock as the master and slave fall out of synchronization (a problem we first noted in Chapter 7).

There are workarounds for driving minilanguages that are not so carefully designed. The prototype for most of them is the Tcl *expect* package. This package assists conversation with CLIs. It's built around the following operation: read from slave until either a given regular-expression pattern is matched or a specified timeout elapses. With this (and, of course, a send-to-slave operation) it's often possible to construct master programs to do reliable dialogues with slave processes even when the latter have not been tailored for the role.

Workalikes of the *expect* package in other languages are available; a Web search for the name of your favorite language with the added keywords "Tcl expect" is quite likely to turn up something useful. As a minilanguage designer, however, you would be unwise to assume that all your users will be *expect* gurus. Even if they are, this is an extra glue layer and a place for things to go wrong.

Be aware of this issue when designing your minilanguage. It may be a good idea to add an option that changes its conversational behavior to make it respond more like an application protocol, with unambiguous end-of-output delimiters and an analog of byte stuffing.

Chapter 9. Generation

Pushing the Specification Level Upwards

The programmer at wit's end ... can often do best by disentangling himself from his code, rearing back, and contemplating his data. Representation is the essence of programming.

--
<author>FredBrooks</author>
The Mythical Man-Month, Anniversary Edition (1975-1995), p. 103

In Chapter 1 we observed that human beings are better at visualizing data than they are at reasoning about control flow. We recapitulate: To see this, compare the expressiveness and explanatory power of a diagram of a fifty-node pointer tree with a flowchart of a fifty-line program. Or (better) of an array initializer expressing a conversion table with an equivalent switch statement. The difference in transparency and clarity is dramatic.[97]

Data is more tractable than program logic. That's true whether the data is an ordinary table, a declarative markup language, a templating system, or a set of macros that will expand to program logic. It's good practice to move as much of the complexity in your design as possible away from procedural code and into data, and good practice to pick data representations that are convenient for humans to maintain and manipulate. Translating those representations into forms that are convenient for machines to process is another job for machines, not for humans.

> Another important advantage of higher-level, more declarative notations is that they lend themselves better to compile-time checking. Procedural notations inherently have complex runtime behavior which is difficult to analyze at compile time. Declarative notations give the implementation much more leverage for finding mistakes, by permitting much more thorough understanding of the intended behavior.
>
> —
> <author>HenrySpencer</author>

These insights ground in theory a set of practices that have always been an important part of the Unix programmer's toolkit — very high-level languages, data-driven programming, code generators, and domain-specific minilanguages. What unifies these is that they are all ways of lifting the generation of code up some levels, so that specifications can be smaller. We've previously noted that defect

[97]For further development of this point see [Bentley].

densities tend to be nearly constant across programming languages; all these practices mean that whatever malign forces generate our bugs will get fewer lines to wreak their havoc on.

In Chapter 8 we discussed the uses of domain-specific minilanguages. In Chapter 14 we'll make the argument for very-high-level languages. In this chapter we'll look at some design studies in data-driven programming and a few examples of ad-hoc code generation; we'll look at some code-generation tools in Chapter 15. As with minilanguages, these methods can enable you to drastically cut the line count of your programs, and correspondingly lower debugging time and maintenance costs.

Data-Driven Programming

When doing *data-driven programming*, one clearly distinguishes code from the data structures on which it acts, and designs both so that one can make changes to the logic of the program by editing not the code but the data structure.

Data-driven programming is sometimes confused with object orientation, another style in which data organization is supposed to be central. There are at least two differences. One is that in data-driven programming, the data is not merely the state of some object, but actually defines the control flow of the program. Where the primary concern in OO is encapsulation, the primary concern in data-driven programming is writing as little fixed code as possible. Unix has a stronger tradition of data-driven programming than of OO.

Programming data-driven style is also sometimes confused with writing state machines. It is in fact possible to express the logic of a state machine as a table or data structure, but hand-coded state machines are usually rigid blocks of code that are far harder to modify than a table.

An important rule when doing any kind of code generation or data-driven programming is this: always push problems upstream. Don't hack the generated code or any intermediate representations by hand — instead, think of a way to improve or replace your translation tool. Otherwise you're likely to find that hand-patching bits which should have been generated correctly by machine will have turned into an infinite time sink.

At the upper end of its complexity scale, data-driven programming merges into writing interpreters for p-code or simple minilanguages of the kind we surveyed in Chapter 8. At other edges, it merges into code generation and state-machine programming. The distinctions are not actually that important; the important part is moving program logic away from hardwired control structures and into data.

Case Study: *ascii*

I maintain a program called *ascii*, a very simple little utility that tries to interpret its command-line arguments as names of ASCII (American Standard Code for Information Interchange) characters and report all the equivalent names. Code and documentation for the tool are available from the project page [http://www.catb.org/~esr/ascii]. Here is an illustrative screenshot:

```
esr@snark:~/WWW/writings/taoup$ ascii 10
ASCII 1/0 is decimal 016, hex 10, octal 020, bits 00010000: called ^P, DLE
Official name: Data Link Escape

ASCII 0/10 is decimal 010, hex 0a, octal 012, bits 00001010: called ^J, LF,↩
NL
Official name: Line Feed
C escape: '\n'
Other names: Newline

ASCII 0/8 is decimal 008, hex 08, octal 010, bits 00001000: called ^H, BS
Official name: Backspace
C escape: '\b'
Other names:

ASCII 0/2 is decimal 002, hex 02, octal 002, bits 00000010: called ^B, STX
Official name: Start of Text
```

One indication that this program was a good idea is the fact that it has an unexpected use — as a quick CLI aid to converting between decimal, hex, octal, and binary representations of bytes.

The main logic of this program could have been coded as a 128-branch case statement. This would, however, have made the code bulky and difficult to maintain. It would also have tangled parts that change relatively rapidly (like the list of slang names for characters) with parts that change slowly or not at all (like the official names), putting them both in the same legend string and making errors during editing much more likely to touch data that ought to be stable.

Instead, we apply data-driven programming. All the character name strings live in a table structure that is quite a bit larger than any of the functions in the code (indeed, counted in lines it is larger

than any *three* of the functions in the program). The code merely navigates the table and does low-level tasks like radix conversions. The initializer actually lives in the file `nametable.h`, which is generated in a way we'll describe later in this chapter.

This organization makes it easy to add new character names, change existing ones, or delete old names by simply editing the table, without disturbing the code.

(The way the program is built is good Unix style, but the output format is questionable. It's hard to see how the output could usefully become the input of any other program, so it does not play well with others.)

Case Study: Statistical Spam Filtering

One interesting case of data-driven programming is statistical learning algorithms for detecting spam (unsolicited bulk email). A whole class of mail filter programs (those easily findable by Web search include *popfile*, *spambayes*, and *bogofilter*) use a database of word correlations to replace the elaborate pattern-matching conditional logic of pattern-matching spam filters.

Programs like these became common on the Internet very rapidly following Paul Graham's landmark paper *A Plan for Spam* [Graham] in 2002. While the explosion was triggered by the increasing cost of the pattern-matching arms race, the statistical-filtering idea was adopted first and fastest by Unix shops. In part, this was certainly because almost all the Internet service providers (who are most burdened by spam, and thus had most incentive to adopt effective new techniques) are Unix shops — but undoubtedly the harmony with some traditional themes in Unix software design helped as well.

Conventional spam filters require that a system administrator, or some other responsible party, maintain information on patterns of text found in spam — names of sites that emit nothing but spam, come-on phrases often used by pornography sites or Internet scam artists, and the like. In his paper, Graham noted accurately that computer programmers like the idea of pattern-matching filters, and sometimes have difficulty seeing past that approach, because it offers them so many opportunities to be clever.

Statistical spam filters, on the other hand, work by collecting feedback about what the user judges to be spam versus nonspam. That feedback is processed into databases of statistical correlation coefficients or *weights* connecting words or phrases to the user's spam/nonspam classification. The most popular algorithms use minor variants of Bayes's Theorem on conditional probabilities, but other techniques (including various sorts of polynomial hashing) are also employed.

In all these programs, the correlation check is a relatively trivial mathematical formula. The weights fed into the formula along with the message being checked serve as implicit control structure for the filtering algorithm.

The problem with conventional pattern-matching spam filters is that they are brittle. Spammers are constantly gaming against the filter-rule databases, forcing the filter maintainers to constantly reprogram their filters to stay ahead in the arms race. Statistical spam filters generate their own filter rules from the user feedback.

In fact, experience with statistical filters seems to show that the particular learning algorithm used is far less important than the quality of the spam and nonspam data sets from which the learning algorithm computes its weights. So the results of statistical filters really are driven more by the shape of the data than by the algorithm.

A Plan for Spam was something of a bombshell because its author argued convincingly that a simple, even crude, statistical approach gave a lower rate of nonspam being erroneously classified as spam than either elaborate pattern-matching techniques or the human eyeball could manage. For Unix programmers, seeing past the lure of clever pattern-matching was far easier than in other programming cultures without as strong an attachment to “Keep It Simple, Stupid!”

Case Study: Metaclass Hacking in *fetchmailconf*

The fetchmailconf(1) dotfile configurator shipped with fetchmail(1) contains an instructive example of advanced data-driven programming in a very high-level, object-oriented language.

In October 1997 a series of questions on the fetchmail-friends mailing list made it clear that end-users were having increasing troubles generating configuration files for *fetchmail*. The file uses a simple, classically-Unixy free-format syntax, but can become forbiddingly complicated when a user has POP3 and IMAP accounts at multiple sites. See Example 9.1 for a somewhat simplified version of the *fetchmail* author’s configuration file.

Example 9.1. Example of fetchmailrc syntax.

```
set postmaster "esr"
set daemon 300

poll imap.ccil.org with proto IMAP and options no dns
   aka snark.thyrsus.com locke.ccil.org ccil.org
```

```
     user esr there is esr here
         options fetchall dropstatus warnings 3600

poll imap.netaxs.com with proto IMAP
     user "esr" there is esr here options dropstatus warnings 3600
```

The design objective of *fetchmailconf* was to completely hide the control file syntax behind a fashionable, ergonomically-correct GUI replete with selection buttons, slider bars and fill-out forms. But the beta design had a problem: it could easily generate configuration files from the user's GUI actions, but could not read and edit existing ones.

The parser for *fetchmail*'s configuration file syntax is rather elaborate. It's actually written in *yacc* and *lex*, the two classic Unix tools for generating language-parsing code in C. For *fetchmailconf* to be able to edit existing configuration files, it at first appeared that it would be necessary to replicate that elaborate parser in fetchmailconf's implementation language — Python.

This tactic seemed doomed. Even leaving aside the amount of duplicative work implied, it is notoriously hard to be certain that two parsers in two different languages accept the same grammar. Keeping them synchronized as the configuration language evolved bid fair to be a maintenance nightmare. It would have violated the SPOT rule we discussed in Chapter 4 wholesale.

This problem stumped me for a while. The insight that cracked it was that *fetchmailconf* could use *fetchmail*'s own parser as a filter! I added a `--configdump` option to *fetchmail* that would parse `.fetchmailrc` and dump the result to standard output in the format of a Python initializer. For the file above, the result would look roughly like Example 9.2 (to save space, some data not relevant to the example is omitted).

Example 9.2. Python structure dump of a *fetchmail* configuration.

```
fetchmailrc = {
   'poll_interval':300,
   "logfile":None,
   "postmaster":"esr",
   'bouncemail':TRUE,
   "properties":None,
   'invisible':FALSE,
   'syslog':FALSE,
   # List of server entries begins here
```

```
'servers': [
   # Entry for site 'imap.ccil.org' begins:
   {
      "pollname":"imap.ccil.org",
      'active':TRUE,
      "via":None,
      "protocol":"IMAP",
      'port':0,
      'timeout':300,
      'dns':FALSE,
      "aka":["snark.thyrsus.com","locke.ccil.org","ccil.org"],
      'users': [
         {
            "remote":"esr",
            "password":"masked_one",
            'localnames':["esr"],
            'fetchall':TRUE,
            'keep':FALSE,
            'flush':FALSE,
            "mda":None,
            'limit':0,
            'warnings':3600,
         }
         ,          ]
   }
   ,
   # Entry for site 'imap.netaxs.com' begins:
   {
      "pollname":"imap.netaxs.com",
      'active':TRUE,
      "via":None,
      "protocol":"IMAP",
      'port':0,
      'timeout':300,
      'dns':TRUE,
      "aka":None,
      'users': [
         {
            "remote":"esr",
            "password":"masked_two",
            'localnames':["esr"],
```

```
                'fetchall':FALSE,
                'keep':FALSE,
                'flush':FALSE,
                "mda":None,
                'limit':0,
                'warnings':3600,
             }
             ,          ]
       }
       ,
    ]
}
```

The major hurdle had been leapt. The Python interpreter could then evaluate the *fetchmail* `--configdump` output and read the configuration available to *fetchmailconf* as the value of the variable ‘fetchmail’.

But this wasn’t quite the last obstacle in the race. What was really needed wasn’t just for *fetchmailconf* to have the existing configuration, but to turn it into a linked tree of live objects. There would be three kinds of objects in this tree: `Configuration` (the top-level object representing the entire configuration), `Site` (representing one of the servers to be polled), and `User` (representing user data attached to a site). The example file describes three site objects, each with one user object attached to it.

The three object classes already existed in *fetchmailconf*. Each had a method that caused it to pop up a GUI edit panel to modify its instance data. The last remaining problem was to somehow transform the static data in this Python initializer into live objects.

I considered writing a glue layer that would explicitly know about the structure of all three classes and use that knowledge to grovel through the initializer creating matching objects, but rejected that idea because new class members were likely to be added over time as the configuration language grew new features. If the object-creation code were written in the obvious way, it would once again be fragile and tend to fall out of synchronization when either the class definitions or the initializer structure dumped by the `--configdump` report generator changed. Again, a recipe for endless bugs.

The better way would be data-driven programming — code that would analyze the shape and members of the initializer, query the class definitions themselves about their members, and then impedance-match the two sets.

Lisp and Java programmers call this *introspection*; in some other object-oriented languages it's called *metaclass hacking* and is generally considered fearsomely esoteric, deep black magic. Most object-oriented languages don't support it at all; in those that do (Perl and Java among them), it tends to be a complicated and fragile undertaking. But Python's facilities for introspection and metaclass hacking are unusually accessible.

See Example 9.3 for the solution code, from near line 1895 of the 1.43 version.

Example 9.3. `copy_instance` metaclass code.

```
def copy_instance(toclass, fromdict):
# Make a class object of given type from a conformant dictionary.
   class_sig = toclass.__dict__.keys(); class_sig.sort()
   dict_keys = fromdict.keys(); dict_keys.sort()
   common = set_intersection(class_sig, dict_keys)
   if 'typemap' in class_sig:
      class_sig.remove('typemap')
   if tuple(class_sig) != tuple(dict_keys):
      print "Conformability error"
#      print "Class signature: " + `class_sig`
#      print "Dictionary keys: " + `dict_keys`
      print "Not matched in class signature: "+ \
                              `set_diff(class_sig, common)`
      print "Not matched in dictionary keys: "+ \
                              `set_diff(dict_keys, common)`
      sys.exit(1)
   else:
      for x in dict_keys:
         setattr(toclass, x, fromdict[x])
```

Most of this code is error-checking against the possibility that the class members and `--configdump` report generation have drifted out of synchronization. It ensures that if the code breaks, the breakage will be detected early — an implementation of the Rule of Repair. The

heart of this function is the last two lines, which set attributes in the class from corresponding members in the dictionary. They're equivalent to this:

```
def copy_instance(toclass, fromdict):
      for x in fromdict.keys():
            setattr(toclass, x, fromdict[x])
```

When your code is this simple, it is far more likely to be right. See Example 9.4 for the code that calls it.

Example 9.4. Calling context for `copy_instance`.

```
# The tricky part - initializing objects from the ‘configuration’
# global.  ‘Configuration’ is the top level of the object tree
# we’re going to mung
Configuration = Controls()
copy_instance(Configuration, configuration)
Configuration.servers = [];
for server in configuration['servers']:
   Newsite = Server()
   copy_instance(Newsite, server)
   Configuration.servers.append(Newsite)
   Newsite.users = [];
   for user in server['users']:
      Newuser = User()
      copy_instance(Newuser, user)
      Newsite.users.append(Newuser)
```

The key point to extract from this code is that it traverses the three levels of the initializer (configuration/server/user), instantiating the correct objects at each level into lists contained in the next object up. Because `copy_instance` is data-driven and completely generic, it can be used on all three levels for three different object types.

This is a new-school sort of example; Python was not even invented until 1990. But it reflects themes that go back to 1969 in the Unix tradition. If meditating on Unix programming as practiced by his predecessors had not taught me constructive laziness — insisting on reuse, and refusing to

write duplicative glue code in accordance with the SPOT rule—I might have rushed into coding a parser in Python. The first key insight that *fetchmail* itself could be made into *fetchmailconf*'s configuration parser might never have happened.

The second insight (that `copy_instance` could be generic) proceeded from the Unix tradition of looking assiduously for ways to avoid hand-hacking. But more specifically, Unix programmers are very used to writing parser specifications to generate parsers for processing language-like markups; from there it was a short step to believing that the rest of the job could be done by some kind of generic tree-walk of the configuration structure. Two separate stages of data-driven programming, one building on the other, were needed to solve the design problem cleanly.

Insights like this can be extraordinarily powerful. The code we have been looking at was written in about ninety minutes, worked the first time it was run, and has been stable in the years since (the only time it has ever broken is when it threw an exception in the presence of genuine version skew). It's less than forty lines and beautifully simple. There is no way that the naïve approach of building an entire second parser could possibly have produced this kind of maintainability, reliability or compactness. Reuse, simplification, generalization, orthogonality: this is the Zen of Unix in action.

In Chapter 10, we'll examine the run-control syntax of *fetchmail* as an example of the standard shell-like metaformat for run-control files. In Chapter 14 we'll use *fetchmailconf* as an example of Python's strength in rapidly building GUIs.

Ad-hoc Code Generation

Unix comes equipped with some powerful special-purpose code generators for purposes like building lexical analyzers (tokenizers) and parsers; we'll survey these in Chapter 15. But there are much simpler, lighter-weight sorts of code generation we can use to make life easier without having to know any compiler theory or write (error-prone) procedural logic.

Here are a couple of simple case studies to illustrate this point:

Case Study: Generating Code for the *ascii* Displays

Called without arguments, *ascii* generates a usage screen that looks like Example 9.5.

Example 9.5. *ascii* usage screen.

```
Usage: ascii [-dxohv] [-t] [char-alias...]
   -t = one-line output  -d = Decimal table  -o = octal table  -x = hex table
   -h = This help screen -v = version information
Prints all aliases of an ASCII character. Args may be chars, C \-escapes,
English names, ^-escapes, ASCII mnemonics, or numerics in
decimal/octal/hex.

Dec Hex    Dec Hex    Dec Hex  Dec Hex  Dec Hex  Dec Hex   Dec Hex   Dec Hex
  0 00 NUL  16 10 DLE  32 20    48 30 0  64 40 @  80 50 P   96 60 ‘  112 70 p
  1 01 SOH  17 11 DC1  33 21 !  49 31 1  65 41 A  81 51 Q   97 61 a  113 71 q
  2 02 STX  18 12 DC2  34 22 "  50 32 2  66 42 B  82 52 R   98 62 b  114 72 r
  3 03 ETX  19 13 DC3  35 23 #  51 33 3  67 43 C  83 53 S   99 63 c  115 73 s
  4 04 EOT  20 14 DC4  36 24 $  52 34 4  68 44 D  84 54 T  100 64 d  116 74 t
  5 05 ENQ  21 15 NAK  37 25 %  53 35 5  69 45 E  85 55 U  101 65 e  117 75 u
  6 06 ACK  22 16 SYN  38 26 &  54 36 6  70 46 F  86 56 V  102 66 f  118 76 v
  7 07 BEL  23 17 ETB  39 27 ’  55 37 7  71 47 G  87 57 W  103 67 g  119 77 w
  8 08 BS   24 18 CAN  40 28 (  56 38 8  72 48 H  88 58 X  104 68 h  120 78 x
  9 09 HT   25 19 EM   41 29 )  57 39 9  73 49 I  89 59 Y  105 69 i  121 79 y
 10 0A LF   26 1A SUB  42 2A *  58 3A :  74 4A J  90 5A Z  106 6A j  122 7A z
 11 0B VT   27 1B ESC  43 2B +  59 3B ;  75 4B K  91 5B [  107 6B k  123 7B {
 12 0C FF   28 1C FS   44 2C ,  60 3C <  76 4C L  92 5C \  108 6C l  124 7C |
 13 0D CR   29 1D GS   45 2D -  61 3D =  77 4D M  93 5D ]  109 6D m  125 7D }
 14 0E SO   30 1E RS   46 2E .  62 3E >  78 4E N  94 5E ^  110 6E n  126 7E ~
 15 0F SI   31 1F US   47 2F /  63 3F ?  79 4F O  95 5F _  111 6F o  127 7F DEL
```

This screen is carefully designed to fit in 23 rows and 79 columns, so that it will fit in a 24×80 terminal window.

This table could be generated at runtime, on the fly. Grinding out the decimal and hex columns would be easy enough. But between wrapping the table at the right places and knowing when to print mnemonics like NUL rather than characters, there would have been enough odd corner cases to make the code distinctly unpleasant. Furthermore, the columns had to be unevenly spaced to make the table fit in 79 columns. But any Unix programmer would reflexively express it as a block of data before finding out these things.

The most naïve way to generate the usage screen would have been to put each line into a C initializer in the `ascii.c` source code, and then have all lines be written out by code that steps through the

initializer. The problem with this method is that the extra data in the C initializer format (trailing newline, string quotes, comma) would make the lines longer than 79 characters, causing them to wrap and making it rather difficult to map the appearance of the code to the appearance of the output. This, in turn, would make the display difficult to edit, which was annoying when I was tinkering it to fit in 24×80 screen cells.

A more sophisticated method using the string-pasting behavior of the ANSI C preprocessor collided with a variant of the same problem. Essentially, any way of inlining the usage screen explicitly would involve punctuation at start and end of line that there's no room for.[98] And copying the table to the screen from a file at runtime seemed like a fragile expedient; after all, the file could get lost.

Here's the solution. The source distribution contains a file that just contains the usage screen, exactly as listed above and named `splashscreen`. The C source contains the following function:

```
void
showHelp(FILE *out, char *progname)
{
  fprintf(out,"Usage: %s [-dxohv] [-t] [char-alias...]\n", progname);
#include "splashscreen.h"

  exit(0);
}
```

And `splashscreen.h` is generated by a makefile production:

```
splashscreen.h: splashscreen
        sed <splashscreen >splashscreen.h \
            -e 's/\\/\\\\/g' -e 's/"/\\"/' -e 's/.*/puts("&");/'
```

So when the program is built, the `splashscreen` file is automatically massaged into a series of output function calls, which are then included by the C preprocessor in the right function.

By generating the code from data, we get to keep the editable version of the usage screen identical to its display appearance. This promotes transparency. Furthermore, we could modify the usage

98 Scripting languages tend to solve this problem more elegantly than C does. Investigate the shell's *here documents* and Python's triple-quote construct to find out how.

screen at will without touching the C code at all, and the right thing would automatically happen on the next build.

For similar reasons, the initializer that holds the name synonym strings is also generated via a *sed* script in the makefile, from a file called `nametable` in the *ascii* source distribution. Most of `nametable` is simply copied into the C initializer. But the generation process would make it easy to adapt this tool for other 8-bit character sets such as the ISO-8859 series (Latin-1 and friends).

This is an almost trivial example, but it nevertheless illustrates the advantages of even simple and ad-hoc code generation. Similar techniques could be applied to larger programs with correspondingly greater benefits.

Case Study: Generating HTML Code for a Tabular List

Let's suppose that we want to put a page of tabular data on a Web page. We want the first few lines to look like Example 9.6.

Example 9.6. Desired output format for the star table.

```
Aalat        David Weber          The Armageddon Inheritance
Aelmos       Alan Dean Foster     The Man who Used the Universe
Aedryr       Steve Miller/Sharon Lee Scout's Progress
Aergistal    Gerard Klein         The Overlords of War
Afdiar       L. Neil Smith        Tom Paine Maru
Agandar      Donald Kingsbury     Psychohistorical Crisis
Aghirnamirr  Jo Clayton           Shadowkill
```

The thick-as-a-plank way to handle this would be to hand-write HTML table code for the desired appearance. Then, each time we want to add a name, we'd have to hand-write another set of <tr> and <td> tags for the entry. This would get very tedious very quickly. But what's worse, changing the format of the list would require hand-hacking every entry.

The superficially clever way to handle this would be to make this data a three-column relation in a database, then use some fancy CGI[99] technique or a database-capable templating engine like PHP to generate the page on the fly. But suppose we know that the list will not change very often, don't

[99] Here, CGI refers not to Computer Graphic Inagery but to the Common Gateway Interface used for live Web content.

want to run a database server just to be able to display this list, and don't want to load the server with unnecessary CGI traffic?

There's a better solution. We put the data in a tabular flat-file format like Example 9.7.

Example 9.7. Master form of the star table.

```
Aalat       :David Weber            :The Armageddon Inheritance
Aelmos      :Alan Dean Foster       :The Man who Used the Universe
Aedryr      :Steve Miller/Sharon Lee :Scout's Progress
Aergistal   :Gerard Klein           :The Overlords of War
Afdiar      :L. Neil Smith          :Tom Paine Maru
Agandar     :Donald Kingsbury       :Psychohistorical Crisis
Aghirnamirr :Jo Clayton             :Shadowkill
```

We could in a pinch have done without the explicit colon field delimiters, using the pattern consisting of two or more spaces as a delimiter, but the explicit delimiter protects us in case we press spacebar twice while editing a field value and fail to notice it.

We then write a script in shell, Perl, Python, or Tcl that massages this file into an HTML table, and run that each time we add an entry. The old-school Unix way would revolve around the following nigh-unreadable sed(1) invocation

```
sed -e 's,^,<tr><td>,' -e 's,$,</td></tr>,' -e 's,:,</td><td>,g'
```

or this perhaps slightly more scrutable awk(1) program:

```
awk -F: '{printf("<tr><td>%s</td><td>%s</td><td>%s</td></tr>\n", \
            $1, $2, $3)}'
```

(If either of these examples interests but mystifies, read the documentation for sed(1) or awk(1). We explained in Chapter 8 that the latter has largely fallen out of use. The former is still an important Unix tool that we haven't examined in detail because (a) Unix programmers already know it, and (b) it's easy for non-Unix programmers to pick up from the manual page once they grasp the basic ideas about pipelines and redirection.)

A new-school solution might center on this Python code, or on equivalent Perl:

```
for row in map(lambda x:x.rstrip().split(':'),sys.stdin.readlines()):
   print "<tr><td>" + "</td><td>".join(row) + "</td></tr>"
```

These scripts took about five minutes each to write and debug, certainly less time than would have been required to either hand-hack the initial HTML or create and verify the database. The combination of the table and this code will be much simpler to maintain than either the under-engineered hand-hacked HTML or the over-engineered database.

A further advantage of this way of solving the problem is that the master file stays easy to search and modify with an ordinary text editor. Another is that we can experiment with different table-to-HTML transformations by tweaking the generator script, or easily make a subset of the report by putting a grep(1) filter before it.

I actually use this technique to maintain the Web page that lists *fetchmail* test sites; the example above is science-fictional only because publishing the real data would reveal account usernames and passwords.

This was a somewhat less trivial example than the previous one. What we've actually designed here is a separation between content and formatting, with the generator script acting as a stylesheet. (This is yet another mechanism-vs.-policy separation.)

The lesson in all these cases is the same. Do as little work as possible. Let the data shape the code. Lean on your tools. Separate mechanism from policy. Expert Unix programmers learn to see possibilities like these quickly and automatically. Constructive laziness is one of the cardinal virtues of the master programmer.

Chapter 10. Configuration

Starting on the Right Foot

Let us watch well our beginnings, and results will manage themselves.
--
<author>AlexanderClark</author>

Under Unix, programs can communicate with their environment in a rich variety of ways. It's convenient to divide these into (a) startup-environment queries and (b) interactive channels. In this chapter, we'll focus primarily on startup-environment queries. The next chapter will discuss interactive channels.

What Should Be Configurable?

Before plunging into the details of different kinds of program configuration, we should ask a high-level question: What things should be configurable?

The gut-level Unix answer is "everything". The Rule of Separation that we discussed in Chapter 1 encourages Unix programmers to build mechanism and defer policy decisions outward toward the user wherever possible. While this tends to produce programs that are powerful and rewarding for expert users, it also tends to produce interfaces that overwhelm novices and casual users with a surfeit of choices, and with configuration files sprouting like weeds.

Unix programmers aren't going to be cured of their tendency to design for their peers and the most sophisticated users any time soon (we'll grapple a bit with the question of whether such a change would actually be desirable in Chapter 20). So it's perhaps more useful to invert the question and ask "What things should *not* be configurable?" Unix practice does offer some guidelines on this.

First, *don't provide configuration switches for what you can reliably detect automatically*. This is a surprisingly common mistake. Instead, look for ways to eliminate configuration switches by autodetection, or by trying alternative methods at runtime until one succeeds. If this strikes you as inelegant or too expensive, ask yourself if you haven't fallen into premature optimization.

> One of the nicest examples of autodetection I experienced was when Dennis Ritchie and I were porting Unix to the Interdata 8/32. This was a big-endian machine, and we had to generate data for that machine on a PDP-11, write a

> magnetic tape, and then load the magnetic tape on the Interdata. A common error was to forget to twiddle the byte order; a checksum error showed you that you had to unmount, remount again on the PDP-11, regenerate the tape, unmount, and remount. Then one day Dennis hacked the Interdata tape reader program so that if it got a checksum error it rewound the tape, toggled 'byte flip' switch and reread it. A second checksum error would kill the load, but 99% of the time it just read the tape and did the right thing. Our productivity shot up, and we pretty much ignored tape byte order from that point on.
>
> —
>
> <author>SteveJohnson</author>

A good rule of thumb is this: Be adaptive unless doing so costs you 0.7 seconds or more of latency. 0.7 seconds is a magic number because, as Jef Raskin discovered while designing the Canon Cat, humans are almost incapable of noticing startup latency shorter than that; it gets lost in the mental overhead of changing the focus of attention.

Second, *users should not see optimization switches*. As a designer, it's *your* job to make the program run economically, not the user's. The marginal gains in performance that a user might collect from optimization switches are usually not worth the interface-complexity cost.

> File-format nonsense (record length, blocking factor, etc) was blessedly eschewed by Unix, but the same kind of thing has roared back in excess configuration goo. KISS became MICAHI: make it complicated and hide it.
>
> —
>
> <author>DougMcIlroy</author>

Finally, *don't do with a configuration switch what can be done with a script wrapper or a trivial pipeline*. Don't put complexity inside your program when you can easily enlist other programs to help get the work done. (Recall our discussion in Chapter 7 of why ls(1) does not have a built-in pager, or an option to invoke it).

Here are some more general questions to consider whenever you find yourself thinking about adding a configuration option:

- Can I leave this feature out? Why am I fattening the manual and burdening the user?

- Could the program's normal behavior be changed in an innocuous way that would make the option unnecessary?
- Is this option merely cosmetic? Should I be thinking less about how to make the user interface configurable and more about how to make it right?
- Should the behavior enabled by this option be a separate program instead?

Proliferating unnecessary options has many bad effects. One of the subtlest but most serious is what it will do to your test coverage.

> Unless it is done very carefully, the addition of an on/off configuration option can lead to a need to double the amount of testing. Since in practice one never does double the amount of testing, the practical effect is to reduce the amount of testing that any given configuration receives. Ten options leads to 1024 times as much testing, and pretty soon you are talking real reliability problems.
>
> —
> <author>SteveJohnson</author>

Where Configurations Live

Classically, a Unix program can look for control information in five places in its startup-time environment:

- Run-control files under `/etc` (or at fixed location elsewhere in systemland).
- System-set environment variables.
- Run-control files (or 'dotfiles') in the user's home directory. (See Chapter 3 for a discussion of this important concept, if it is unfamiliar.)
- User-set environment variables.
- Switches and arguments passed to the program on the command line that invoked it.

These queries are usually done in the order listed above. That way, later (more local) settings override earlier (more global) ones. Settings found earlier can help the program compute locations for later retrievals of configuration data.

When thinking about which mechanism to use to pass configuration data to a program, bear in mind that good Unix practice demands using whichever one most closely matches the expected lifetime of the preference. Thus: for preferences which are very likely to change between invocations, use command-line switches. For preferences which change seldom, but that should be under individual user control, use a run-control file in the user's home directory. For preference information that needs to be set site-wide by a system administrator and *not* changed by users, use a run-control file in system space.

We'll discuss each of these places in more detail, then examine some case studies.

Run-Control Files

A run-control file is a file of declarations or commands associated with a program that it interprets on startup. If a program has site-specific configuration shared by all users at a site, it will often have a run-control file under the `/etc` directory. (Some Unixes have an `/etc/conf` subdirectory that collects such data.)

User-specific configuration information is often carried in a hidden run-control file in the user's home directory. Such files are often called 'dotfiles' because they exploit the Unix convention that a filename beginning with a dot is normally invisible to directory-listing tools.[100]

Programs can also have run-control or dot directories. These group together several configuration files that are related to the program, but that are most conveniently treated separately (perhaps because they relate to different subsystems of the program, or have differing syntaxes).

Whether file or directory, convention now dictates that the location of the run-control information has the same basename as the executable that reads it. An older convention still common among system programs uses the executable's name with the suffix 'rc' for 'run control'.[101] Thus, if you write a program called 'seekstuff' that has both site-wide and user-specific configuration, an experienced Unix user would expect to find the former at `/etc/seekstuff` and the latter at `.seekstuff` in the

[100] To make dotfiles visible, use the `-a` option of ls(1).
[101] The 'rc' suffix goes back to Unix's grandparent, CTSS. It had a command-script feature called "runcom". Early Unixes used 'rc' for the name of the operating system's boot script, as a tribute to CTSS runcom.

user's home directory; but it would be unsurprising if the locations were `/etc/seekstuffrc` and `.seekstuffrc`, especially if seekstuff were a system utility of some sort.

In Chapter 5 we described a somewhat different set of design rules for textual data-file formats, and discussed how to optimize for different weightings of interoperability, transparency and transaction economy. Run-control files are typically only read once at program startup and not written; economy is therefore usually not a major concern. Interoperability and transparency both push us toward textual formats designed to be read by human beings and modified with an ordinary text editor.

While the semantics of run-control files are of course completely program dependent, there are some design rules about run-control syntax that are widely observed. We'll describe those next; but first we'll describe an important exception.

If the program is an interpreter for a language, then it is expected to be simply a file of commands in the syntax of that language, to be executed at startup. This is an important rule, because Unix tradition strongly encourages the design of all kinds of programs as special-purpose languages and minilanguages. Well-known examples with dotfiles of this kind include the various Unix command shells and the *Emacs* programmable editor.

(One reason for this design rule is the belief that special cases are bad news — thus, that any switch that changes the behavior of a language should be settable from within the language. If as a language designer you find that you *cannot* express all the startup settings of a language in the the language itself, a Unix programmer would say you have a design problem — which is what you should be fixing, rather than devising a special-case run-control syntax.)

This exception aside, here are the normal style rules for run-control syntaxes. Historically, they are patterned on the syntax of Unix shells:

1. *Support explanatory comments, and lead them with #.* The syntax should also ignore whitespace before #, so that comments on the same line as configuration directives are supported.

2. *Don't make insidious whitespace distinctions.* That is, treat runs of spaces and tabs, syntactically the same as a single space. If your directive format is line-oriented, it is good form to ignore trailing spaces and tabs on lines. The metarule is that the interpretation of the file should not depend on distinctions a human eye can't see.

3. *Treat multiple blank lines and comment lines as a single blank line.* If the input format uses blank lines as separators between records, you probably want to ensure that a comment line does not end a record.

4. *Lexically treat the file as a simple sequence of whitespace-separated tokens, or lines of tokens.* Complicated lexical rules are hard to learn, hard to remember, and hard for humans to parse. Avoid them.

5. *But, support a string syntax for tokens with embedded whitespace.* Use single quote or double quote as balanced delimiters. If you support both, beware of giving them different semantics as they have in shell; this is a well-known source of confusion.

6. *Support a backslash syntax for embedding unprintable and special characters in strings.* The standard pattern for this is the backslash-escape syntax supported by C compilers. Thus, for example, it would be quite surprising if the string `"a\tb"` were not interpreted as a character 'a', followed by a tab, followed by the character 'b'.

Some aspects of shell syntax, on the other hand, should *not* be emulated in run-control syntaxes — at least not without a good and specific reason. The shell's baroque quoting and bracketing rules, and its special metacharacters for wildcards and variable substitution, both fall into this category.

It bears repeating that the point of these conventions is to reduce the amount of novelty that users have to cope with when they read and edit the run-control file for a program they have never seen before. Therefore, if you have to break the conventions, try to do so in a way that makes it visually obvious that you have done so, document your syntax with particular care, and (most importantly) design it so it's easy to pick up by example.

These standard style rules only describe conventions about tokenizing and comments. The names of run-control files, their higher-level syntax, and the semantic interpretation of the syntax are usually application-specific. There are a very few exceptions to this rule, however; one is dotfiles which have become 'well-known' in the sense that they routinely carry information used by a whole class of applications. Sharing run-control file formats in this way reduces the amount of novelty users have to cope with.

Of these, probably the best established is the `.netrc` file. Internet client programs that must track host/password pairs for a user can usually get them from the `.netrc` file, if it exists.

Case Study: The `.netrc` File

The `.netrc` file is a good example of the standard rules in action. An example, with the passwords changed to protect the innocent, is in Example 10.1.

Example 10.1. A `.netrc` example.

```
# FTP access to my Web host
machine unix1.netaxs.com
      login esr
      password joesatriani

# My main mailserver at Netaxs
machine imap.netaxs.com
      login esr
      password jeffbeck

# Auxiliary IMAP maildrop at CCIL
machine imap.ccil.org
   login esr
   password marcbonilla

# Auxiliary POP maildrop at CCIL
machine pop3.ccil.org
   login esr
   password ericjohnson

# Shell account at CCIL
machine locke.ccil.org
   login esr
   password stevemorse
```

Observe that this format is easy to parse by eyeball even if you've never seen it before; it's a set of machine/login/password triples, each of which describes an account on a remote host. This kind of transparency is important — much more important, actually, than the time economy of faster interpretation or the space economy of a more compact and cryptic file format. It economizes the far more valuable resource that is *human* time, by making it likely that a human being will be able to read and modify the format without having to read a manual or use a tool less familiar than a plain old text editor.

Observe also that this format is used to supply information for multiple services — an advantage, because it means sensitive password information need only be stored in one place. The `.netrc` format was designed for the original Unix FTP client program. It's used by all FTP clients, and also understood by some telnet clients, and by the smbclient(1) command-line tool, and by the *fetchmail* program. If you are writing an Internet client that must do password authentication through remote logins, the Rule of Least Surprise demands that it use the contents of `.netrc` as defaults.

Portability to Other Operating Systems

Systemwide run-control files are a design tactic that can be used on almost any operating system, but dotfiles are rather more difficult to map to a non-Unix environment. The critical thing missing from most non-Unix operating systems is true multiuser capability and the notion of a per-user home directory. DOS and Windows versions up to ME (including 95 and 98), for example, completely lack any such notion; all configuration information has to be stored either in systemwide run-control files at a fixed location, the Windows registry, or configuration files in the same directory a program is run from. Windows NT has some notion of per-user home directories (which made its way into Windows 2000 and XP), but it is only poorly supported by the system tools.

Environment Variables

When a Unix program starts up, the environment accessible to it includes a set of name to value associations (names and values are both strings). Some of these are set manually by the user; others are set by the system at login time, or by your shell or terminal emulator (if you're running one). Under Unix, environment variables tend to carry information about file search paths, system defaults, the current user ID and process number, and other key bits of information about the runtime einvironment of programs. At a shell prompt, typing **set** followed by a newline will list all currently defined shell variables.

In C and C++ these values can be queried with the library function getenv(3). Perl and Python initialize environment-dictionary objects at startup. Other languages generally follow one of these two models.

System Environment Variables

There are a number of well-known environment variables you can expect to find defined on startup of a program from the Unix shell. These (especially `HOME`) will often need to be evaluated *before* you read a local dotfile.

`USER` Login name of the account under which this session is logged in (BSD convention).

`LOGNAME` Login name of the account under which this session is logged in (System V convention).

`HOME` Home directory of the user running this session.

`COLUMNS` The number of character-cell columns on the controlling terminal or terminal-emulator window.

`LINES` The number of character-cell rows on the controlling terminal or terminal-emulator window.

`SHELL` The name of the user's command shell (often used by shellout commands).

`PATH` The list of directories that the shell searches when looking for executable commands to match a name.

`TERM` Name of the terminal type of the session console or terminal emulator window (see the terminfo case study in Chapter 6 for background). TERM is special in that programs to create remote sessions over the network (such as *telnet* and *ssh*) are expected to pass it through and set it in the remote session.

(This list is representative, but not exhaustive.)

The `HOME` variable is especially important, because many programs use it to find the calling user's dotfiles (others call some functions in the C runtime library to get the calling user's home directory).

Note that some or all of these system environment variables may *not* be set when a program is started by some other method than a shell spawn. In particular, daemon listeners on a TCP/IP socket often don't have these variables set — and if they do, the values are unlikely to be useful.

Finally, note that there is a tradition (exemplified by the `PATH` variable) of using a colon as a separator when an environment variable must contain multiple fields, especially when the fields can be interpreted as a search path of some sort. Note that some shells (notably *bash* and *ksh*) *always* interpret colon-separated fields in an environment variable as filenames, which means in particular that they expand ~ in these fields to the user's home directory.

User Environment Variables

Although applications are free to interpret environment variables outside the system-defined set, it is nowadays fairly unusual to actually do so. Environment values are not really suitable for passing structured information into a program (though it can in principle be done via parsing of the values). Instead, modern Unix applications tend to use run-control files and dotfiles.

There are, however, some design patterns in which user-defined environment variables can be useful:

Application-independent preferences that need to be shared by a large number of different programs. This set of 'standard' preferences changes only slowly, because lots of different programs need to recognize each one before it becomes useful.[102] Here are the standard ones:

`EDITOR`
: The name of the user's preferred editor (often used by shellout commands).[103]

`MAILER`
: The name of the user's preferred mail user agent (often used by shellout commands).

`PAGER`
: The name of the user's preferred program for browsing plaintext.

`BROWSER`
: The name of the user's preferred program for browsing Web URLs. This one, as of 2003, is still very new and not yet widely implemented.

When to Use Environment Variables

What both user and system environment variables have in common is that it would be annoying to have to replicate the information they contain in a large number of application run-control files, and extremely annoying to have to change that information everywhere when your preference changes. Typically, the user sets these variables in his or her shell session startup file.

A value varies across several contexts that share dotfiles, or a parent needs to pass information to multiple child processes. Some pieces of start-up information are expected to vary across several contexts in which the calling user would share common run-control files and dotfiles. For a system-level example, consider several shell sessions open through terminal emulator windows on an X

[102] Nobody knows a really graceful way to represent this sort of distributed preference data; environment variables probably are not it, but all the known alternatives have equally nasty problems.

[103] Actually, most Unix programs first check `VISUAL`, and only if that's not set will they consult `EDITOR`. That's a relic from the days when people had different preferences for line-oriented editors and visual editors.

desktop. They will all see the same dotfiles, but might have different values of `COLUMNS`, `LINES`, and `TERM`. (Old-school shell programming used this method extensively; makefiles still do.)

A value varies too often for dotfiles, but doesn't change on every startup. A user-defined environment variable may (for example) be used to pass a file system or Internet location that is the root of a tree of files that the program should play with. The CVS version-control system interprets the variable `CVSROOT` this way, for example. Several newsreader clients that fetch news from servers using the NNTP protocol interpret the variable `NNTPSERVER` as the location of the server to query.

A process-unique override needs to be expressed in a way that doesn't require the command-line invocation to be changed. A user-defined environment variable can be useful for situations in which, for whatever reason, it would be inconvenient to have to change an application dotfile or supply command-line options (perhaps it is expected that the application will normally be used inside a shell wrapper or within a makefile). A particularly important context for this sort of use is debugging. Under Linux, for example, manipulating the variable `LD_LIBRARY_PATH` associated with the ld(1) linking loader enables you to change where libraries are loaded from — perhaps to pick up versions that do buffer-overflow checking or profiling.

In general, a user-defined environment variable can be an effective design choice when the value changes often enough to make editing a dotfile each time inconvenient, but not necessarily every time (so always setting the location with a command-line option would also be inconvenient). Such variables should typically be evaluated *after* a local dotfile and be permitted to override settings in it.

There is one traditional Unix design pattern that we do not recommend for new programs. Sometimes, user-set environment variables are used as a lightweight substitute for expressing a program preference in a run-control file. The venerable nethack(1) dungeon-crawling game, for example, reads a `NETHACKOPTIONS` environment variable for user preferences. This is an old-school technique; modern practice would lean toward parsing them from a `.nethack` or `.nethackrc` run-control file.

The problem with the older style is that it makes tracking where your preference information lives more difficult than it would be if you knew the program had a run-control file under your home directory. Environment variables can be set anywhere in several different shell run-control files — under Linux these are likely to include `.profile`, `.bash_profile`, and `.bashrc` at least. These files are cluttered and fragile things, so as the code overhead of having an option-parser has come to seem less significant preference information has tended to migrate out of environment variables into dotfiles.

Portability to Other Operating Systems

Environment variables have only very limited portability off Unix. Microsoft operating systems have an environment-variable feature modeled on that of Unix, and use a `PATH` variable as Unix does to set the binary search path, but most of other variables that Unix shell programmers take for granted (such as process ID or current working directory) are not supported. Other operating systems (including classic MacOS) generally do not have any local equivalent of environment variables.

Command-Line Options

Unix tradition encourages the use of command-line switches to control programs, so that options can be specified from scripts. This is especially important for programs that function as pipes or filters. Three conventions for how to distinguish command-line options from ordinary arguments exist; the original Unix style, the GNU style, and the X toolkit style.

In the original Unix tradition, command-line options are single letters preceded by a single hyphen. Mode-flag options that do not take following arguments can be ganged together; thus, if `-a` and `-b` are mode options, `-ab` or `-ba` is also correct and enables both. The argument to an option, if any, follows it (optionally separated by whitespace). In this style, lowercase options are preferred to uppercase. When you use uppercase options, it's good form for them to be special variants of the lowercase option.

The original Unix style evolved on slow ASR-33 teletypes that made terseness a virtue; thus the single-letter options. Holding down the shift key required actual effort; thus the preference for lower case, and the use of "-" (rather than the perhaps more logical "+") to enable options.

The GNU style uses option keywords (rather than keyword letters) preceded by two hyphens. It evolved years later when some of the rather elaborate GNU utilities began to run out of single-letter option keys (this constituted a patch for the symptom, not a cure for the underlying disease). It remains popular because GNU options are easier to read than the alphabet soup of older styles. GNU-style options cannot be ganged together without separating whitespace. An option argument (if any) can be separated by either whitespace or a single "=" (equal sign) character.

The GNU double-hyphen option leader was chosen so that traditional single-letter options and GNU-style keyword options could be unambiguously mixed on the same command line. Thus, if your initial design has few and simple options, you can use the Unix style without worrying about causing an incompatible 'flag day' if you need to switch to GNU style later on. On the other hand, if you

are using the GNU style, it is good practice to support single-letter equivalents for at least the most common options.

The X toolkit style, confusingly, uses a single hyphen and keyword options. It is interpreted by X toolkits that filter out and process certain options (such as `-geometry` and `-display`) before handing the filtered command line to the application logic for interpretation. The X toolkit style is not properly compatible with either the classic Unix or GNU styles, and should not be used in new programs unless the value of being compatible with older X conventions seems very high.

Many tools accept a bare hyphen, not associated with any option letter, as a pseudo-filename directing the application to read from standard input. It is also conventional to recognize a double hyphen as a signal to stop option interpretation and treat all following arguments literally.

Most Unix programming languages offer libraries that will parse a command line for you in either classic-Unix or GNU style (interpreting the double-hyphen convention as well).

The `-a` to `-z` of Command-Line Options

Over time, frequently-used options in well-known Unix programs have established a loose sort of semantic standard for what various flags might be expected to mean. The following is a list of options and meanings that should prove usefully unsurprising to an experienced Unix user:

`-a`
: All (without argument). If there is a GNU-style `--all` option, for `-a` to be anything but a synonym for it would be quite surprising. Examples: fuser(1), fetchmail(1).

: Append, as in tar(1). This is often paired with `-d` for delete.

`-b`
: Buffer or block size (with argument). Set a critical buffer size, or (in a program having to do with archiving or managing storage media) set a block size. Examples: du(1), df(1), tar(1).

: Batch. If the program is naturally interactive, `-b` may be used to suppress prompts or set other options appropriate to accepting input from a file rather than a human operator. Example: flex(1).

`-c` Command (with argument). If the program is an interpreter that normally takes commands from standard input, it is expected that the option of a `-c` argument will be passed to it as a single line of input. This convention is particularly strong for shells and shell-like interpreters. Examples: sh(1), ash(1), bsh(1), ksh(1), python(1). Compare `-e` below.

Check (without argument). Check the correctness of the file argument(s) to the command, but don't actually perform normal processing. Frequently used as a syntax-check option by programs that do interpretation of command files. Examples: getty(1), perl(1).

`-d` Debug (with or without argument). Set the level of debugging messages. This one is very common.

Occasionally `-d` has the sense of 'delete' or 'directory'.

`-D` Define (with argument). Set the value of some symbol in an interpreter, compiler, or (especially) macro-processor-like application. The model is the use of `-D` by the C compiler's macro preprocessor. This is a strong association for most Unix programmers; don't try to fight it.

`-e` Execute (with argument). Programs that are wrappers, or that can be used as wrappers, often allow `-e` to set the program they hand off control to. Examples: xterm(1), perl(1).

Edit. A program that can open a resource in either a read-only or editable mode may allow `-e` to specify opening in the editable mode. Examples: crontab(1), and the get(1) utility of the SCCS version-control system.

Occasionally `-e` has the sense of 'exclude' or 'expression'.

`-f` — File (with argument). Very often used with an argument to specify an input (or, less frequently, output) file for programs that need to randomly access their input or output (so that redirection via < or > won't suffice). The classic example is tar(1); others abound. It is also used to indicate that arguments normally taken from the command line should be taken from a file instead; see awk(1) and egrep(1) for classic examples. Compare `-o` below; often, `-f` is the input-side analog of `-o`.

Force (typically without argument). Force some operation (such as a file lock or unlock) that is normally performed conditionally. This is less common.

Daemons often use `-f` in a way that combines these two meanings, to force processing of a configuration file from a nondefault location. Examples: ssh(1), httpd(1), and many other daemons.

`-h` — Headers (typically without argument). Enable, suppress, or modify headers on a tabular report generated by the program. Examples: pr(1), ps(1).

Help. This is actually less common than one might expect offhand — for much of Unix's early history developers tended to think of on-line help as memory-footprint overhead they couldn't afford. Instead they wrote manual pages (this shaped the man-page style in ways we'll discuss in Chapter 18).

`-i` — Initialize (usually without argument). Set some critical resource or database associated with the program to an initial or empty state. Example: ci(1) in RCS.

Interactive (usually without argument). Force a program that does not normally query for confirmation to do so. There are classical examples (rm(1), mv(1)) but this use is not common.

`-I` Include (with argument). Add a file or directory name to those searched for resources by the application. All Unix compilers with any equivalent of source-file inclusion in their languages use `-I` in this sense. It would be extremely surprising to see this option letter used in any other way.

`-k` Keep (without argument). Suppress the normal deletion of some file, message, or resource. Examples: passwd(1), bzip(1), and fetchmail(1).

Occasionally `-k` has the sense of 'kill'.

`-l` List (without argument). If the program is an archiver or interpreter/player for some kind of directory or archive format, it would be quite surprising for `-l` to do anything but request an item listing. Examples: arc(1), binhex(1), unzip(1). (However, tar(1) and cpio(1) are exceptions.)

In programs that are already report generators, `-l` almost invariably means "long" and triggers some kind of long-format display revealing more detail than the default mode. Examples: ls(1), ps(1).

Load (with argument). If the program is a linker or a language interpreter, `-l` invariably loads a library, in some appropriate sense. Examples: gcc(1), f77(1), emacs(1).
D1Login. In programs such as rlogin(1) and ssh(1) that need to specify a network identity, `-l` is how you do it.

Occasionally `-l` has the sense of 'length' or 'lock'.

`-m` Message (with argument). Used with an argument, `-m` passes it in as a message string for some logging or announcement purpose. Examples: ci(1), cvs(1).

Occasionally `-m` has the sense of 'mail', 'mode', or 'modification-time'.

`-n`
: Number (with argument). Used, for example, for page number ranges in programs such as head(1), tail(1), nroff(1), and troff(1). Some networking tools that normally display DNS names accept `-n` as an option that causes them to display the raw IP addresses instead; ifconfig(1) and tcpdump(1) are the archetypal examples.

  Not (without argument). Used to suppress normal actions in programs such as make(1).

`-o`
: Output (with argument). When a program needs to specify an output file or device by name on the command line, the `-o` option does it. Examples: as(1), cc(1), sort(1). On anything with a compiler-like interface, it would be extremely surprising to see this option used in any other way. Programs that support `-o` often (like gcc) have logic that allows it to be recognized after ordinary arguments as well as before.

`-p`
: Port (with argument). Especially used for options that specify TCP/IP port numbers. Examples: cvs(1), the PostgreSQL tools, the smbclient(1), snmpd(1), ssh(1).

  Protocol (with argument). Examples: fetchmail(1), snmpnetstat(1).

`-q`
: Quiet (usually without argument). Suppress normal result or diagnostic output. This is very common. Examples: ci(1), co(1), make(1). See also the 'silent' sense of `-s`.

`-r` (also `-R`)
: Recurse (without argument). If the program operates on a directory, then this option might tell it to recurse on all subdirectories. Any other use in a utility that operated on directories would be quite surprising. The classic example is, of course, cp(1).

  Reverse (without argument). Examples: ls(1), sort(1). A filter might use this to reverse its normal translation action (compare `-d`).

`-s`
: Silent (without argument). Suppress normal diagnostic or result output (similar to `-q`; when both are supported, `q` means 'quiet' but `-s` means 'utterly silent'). Examples: csplit(1), ex(1), fetchmail(1).

  Subject (with argument). *Always* used with this meaning on commands that send or manipulate mail or news messages. It is extremely important to support this, as programs that send mail expect it. Examples: mail(1), elm(1), mutt(1).

  Occasionally `-s` has the sense of 'size'.

`-t`
: Tag (with argument). Name a location or give a string for a program to use as a retrieval key. Especially used with text editors and viewers. Examples: cvs(1), ex(1), less(1), vi(1).

`-u`
: User (with argument). Specify a user, by name or numeric UID. Examples: crontab(1), emacs(1), fetchmail(1), fuser(1), ps(1).

`-v`
: Verbose (with or without argument). Used to enable transaction-monitoring, more voluminous listings, or debugging output. Examples: cat(1), cp(1), flex(1), tar(1), many others.

  Version (without argument). Display program's version on standard output and exit. Examples: cvs(1), chattr(1), patch(1), uucp(1). More usually this action is invoked by `-V`.

`-V`
: Version (without argument). Display program's version on standard output and exit (often also prints compiled-in configuration details as well). Examples: gcc(1), flex(1), hostname(1), many others. It would be quite surprising for this switch to be used in any other way.

`-w`
: Width (with argument). Especially used for specifying widths in output formats. Examples: faces(1), grops(1), od(1), pr(1), shar(1).

  Warning (without argument). Enable warning diagnostics, or suppress them. Examples: fetchmail(1), flex(1), nsgmls(1).

`-x` Enable debugging (with or without argument). Like `-d`. Examples: sh(1), uucp(1).

Extract (with argument). List files to be extracted from an archive or working set. Examples: tar(1), zip(1).

`-y` Yes (without argument). Authorize potentially destructive actions for which the program would normally require confirmation. Examples: fsck(1), rz(1).

`-z` Enable compression (without argument). Archiving and backup programs often use this. Examples: bzip(1), GNU tar(1), zcat(1), zip(1), cvs(1).

The preceding examples are taken from the Linux toolset, but should be good on most modern Unixes.

When you're choosing command-line option letters for your program, look at the manual pages for similar tools. Try to use the same option letters they use for the analogous functions of your program. Note that some particular application areas that have particularly strong conventions about command-line switches which you violate at your peril — compilers, mailers, text filters, network utilities and X software are all notable for this. Anybody who wrote a mail agent that used `-s` as anything but a Subject switch, for example, would have scorn rightly heaped upon the choice.

The GNU project recommends conventional meanings for a few double-dash options in the GNU coding standards.[104] It also lists long options which, though not standardized, are used in many GNU programs. If you are using GNU-style options, and some option you need has a function similar to one of those listed, by all means obey the Rule of Least Surprise and reuse the name.

Portability to Other Operating Systems

To have command-line options, you have to have a command line. The MS-DOS family does, of course, though in Windows it's hidden by a GUI and its use is discouraged; the fact that the option character is normally '`/`' rather than '`-`' is merely a detail. MacOS classic and other pure GUI environments have no close equivalent of command-line options.

[104] See the Gnu Coding Standards [http://www.gnu.org/prep/standards.html].

How to Choose among the Methods

We've looked in turn at system and user run-control files, at environment variables, and at command-line arguments. Observe the progression from least easily changed to most easily changed. There is a strong convention that well-behaved Unix programs that use more than one of these places should look at them in the order given, allowing later settings to override earlier ones (there are specific exceptions, such as command-line options that specify where a dotfile should be found).

In particular, environment settings usually override dotfile settings, but can be overridden by command-line options. It is good practice to provide a command-line option like the `-e` of make(1) that can override environment settings or declarations in run-control files; that way the program can be scripted with well-defined behavior regardless of the way the run-control files look or environment variables are set.

Which of these places you choose to look at depends on how much persistent configuration state your program needs to keep around between invocations. Programs designed mainly to be used in a batch mode (as generators or filters in pipelines, for example) are usually completely configured with command-line options. Good examples of this pattern include ls(1), grep(1) and sort(1). At the other extreme, large programs with complicated interactive behavior may rely entirely on run-control files and environment variables, and normal use involves few command-line options or none at all. Most X window managers are a good example of this pattern.

(Unix has the capability for the same file to have multiple names or 'links'. At startup time, every program has available to it the filename through which it was called. One other way to signal to a program that has several modes of operation which one it should come up in is to give it a link for each mode, have it find out which link it was called through, and change its behavior accordingly. But this technique is generally considered unclean and seldom used.)

Let's look at a couple of programs that gather configuration data from all three places. It will be instructive to consider why, for each given piece of configuration data, it is collected as it is.

Case Study: *fetchmail*

The *fetchmail* program uses only two environment variables, `USER` and `HOME`. These variables are in the predefined set initialized by the system; many programs use them.

The value of `HOME` is used to find the dotfile `.fetchmailrc`, which contains configuration information in a fairly elaborate syntax obeying the shell-like lexical rules described above. This

is appropriate because, once it has been initially set up, Fetchmail's configuration will change only infrequently.

There is neither an `/etc/fetchmailrc` nor any other systemwide file specific to fetchmail. Normally such files hold configuration that's not specific to an individual user. fetchmail does use a small set of properties with this kind of scope — specifically, the name of the local postmaster, and a few switches and values describing the local mail transport setup (such as the port number of the local SMTP listener). In practice, however, these are seldom changed from their compiled-in default values. When they are changed, they tend to be modified in user-specific ways. Thus, there has been no demand for a systemwide fetchmail run-control file.

Fetchmail can retrieve host/login/password triples from a `.netrc` file. Thus, it gets authenticator information in the least surprising way.

Fetchmail has an elaborate set of command-line options, which nearly but do not entirely replicate what the `.fetchmailrc` can express. The set was not originally large, but grew over time as new constructs were added to the `.fetchmailrc` minilanguage and parallel command-line options for them were added more or less reflexively.

The intent of supporting all these options was to make fetchmail easier to script by allowing users to override bits of its run control from the command line. But it turns out that outside of a few options like `--fetchall` and `--verbose` there is little demand for this — and none that can't be satisfied with a shellscript that creates a temporary run-control file on the fly and then feeds it to fetchmail using the `-f` option.

Thus, most of the command-line options are never used, and in retrospect including them was probably a mistake; they bulk up the fetchmail code a bit without accomplishing anything very useful.

> If bulking up the code were the only problem, nobody would care, except for a couple of maintainers. However, options increase the chances of error in code, particularly due to unforeseen interactions among rarely used options. Worse, they bulk up the manual, which is a burden on everybody.
>
> —
>
> <author>DougMcIlroy</author>

There is a lesson here; had I thought carefully enough about fetchmail's usage pattern and been a little less ad-hoc about adding features, the extra complexity might have been avoided.

> An alternative way of dealing with such situations, which doesn't clutter up either the code or the manual much, is to have a "set option variable" option, such as the `-O` option of *sendmail*, which lets you specify an option name and value, and sets that name to that value as if such a setting had been given in a configuration file. A more powerful variant of this is what *ssh* does with its `-o` option: the argument to `-o` is treated as if it were a line appended to the configuration file, with the full config-file syntax available. Either of these approaches gives people with unusual requirements a way to override configuration from the command line, without requiring you to provide a separate option for each bit of configuration that might be overridden.
>
> —
> <author>HenrySpencer</author>

Case Study: The XFree86 Server

The X windowing system is the engine that supports bitmapped displays on Unix machines. Unix applications running through a client machine with a bitmapped display get their input events through X and send screen-painting requests to it. Confusingly, X 'servers' actually run on the client machine — they exist to serve requests to interact with the client machine's display device. The applications sending those requests to the X server are called 'X clients', even though they may be running on a server machine. And no, there is no way to explain this inverted terminology that is not confusing.

X servers have a forbiddingly complex interface to their environment. This is not surprising, as they have to deal with a wide range of complex hardware and user preferences. The environment queries common to all X servers, documented on the X(1) and Xserver(1) pages, therefore make a useful example for study. The implementation we examine here is XFree86, the X implementation used under Linux and several other open-source Unixes.

At startup, the XFree86 server examines a systemwide run-control file; the exact pathname varies between X builds on different platforms, but the basename is XF86Config. The XF86Config file has a shell-like syntax as described above. Example 10.2 is a sample section of an XF86Config file.

Example 10.2. X configuration example.

```
# The 16-color VGA server
```

```
Section "Screen"
   Driver     "vga16"
   Device     "Generic VGA"
   Monitor    "LCD Panel 1024x768"
   Subsection  "Display"
      Modes     "640x480" "800x600"
      ViewPort   0 0
   EndSubsection
EndSection
```

The XF86Config file describes the host machine's display hardware (graphics card, monitor), keyboard, and pointing device (mouse/trackball/glidepad). It's appropriate for this information to live in a systemwide run-control file, because it applies to all users of the machine.

Once X has acquired its hardware configuration from the run control file, it uses the value of the environment variable `HOME` to find two dotfiles in the calling user's home directory. These files are `.Xdefaults` and `.xinitrc`.[105]

The `.Xdefaults` file specifies per-user, application-specific resources relevant to X (trivial examples of these might include font and foreground/background colors for a terminal emulator). The phrase 'relevant to X' indicates a design problem, however. Collecting all these resource declarations in one place is convenient for inspecting and editing them, but it is not always clear what should be declared in `.Xdefaults` and what belongs in an application-specific dotfile. The `.xinitrc` file specifies the commands that should be run to initialize the user's X desktop just after server startup. These programs will almost always include a window or session manager.

X servers have a large set of command-line options. Some of these, such as the `-fp` (font path) option, override the XF86Config. Some are intended to help track server bugs, such as the `-audit` option; if these are used at all, they are likely to vary quite frequently between test runs and are therefore poor candidates to be included in a run-control file. A very important option is the one that sets the server's display number. Multiple servers may run on a host provided each has a unique display number, but all instances share the same run-control file(s); thus, the display number cannot be derived solely from those files.

On Breaking These Rules

[105] The `.xinitrc` is analogous to a Startup folder on Windows and other operating systems.

The conventions described in this chapter are not absolute, but violating them will increase friction costs for users and developers in the future. Break them if you must — but be sure you know exactly why you are doing so before you do it. And if you do break them, make sure that attempts to do things in conventional ways break noisily, giving proper error feedback in accordance with the Rule of Repair.

Chapter 11. Interfaces

User-Interface Design Patterns in the Unix Environment

All our knowledge has its origins in our perceptions.
--
<author>LeonardoDa Vinci</author>

The interface of a program is the sum of all the ways that it communicates with human users and other programs. In Chapter 10, we discussed the use of environment variables, switches, run-control files and other parts of start-up-time interfaces. In this chapter, we'll untangle the history and explain the pragmatics of Unix interfaces after startup time. Because user-interface code normally consumes 40% or more of development time, knowing good design patterns is especially important here in order to avoid a lot of false starts and time-intensive rewrites.

In the Unix tradition of interface design, we encounter two themes over and over again. One is anticipatory design for communication with other programs; the other is the Rule of Least Surprise.

Unix programs can give you extra power from being used in synergistic combinations; we discussed various methods for hooking together such combinations in Chapter 7. The 'other programs' part of Unix interface design is not an afterthought or a marginal case as it is under many other operating systems. Rather, it is a central challenge that has to be balanced and integrated carefully with the demands of interface design for human users.

Much of Unix-community tradition about program interface design may seem odd and arbitrary — or even, in the age of the GUI, downright regressive — when you encounter that tradition for the first time. But in spite of various blemishes and irregularities, that tradition has an inner logic to it which is worth learning and understanding. It reflects heuristics accumulated over Unix's long history about ways to do effective communication both with human beings and with other programs. And it includes a set of conventions which create commonalities between programs — it defines 'least surprising' alternatives for a wide range of common interface-design problems.

After startup, programs normally get input or commands from the following sources:

- Data and commands presented on the program's standard input.

- Inputs passed through IPC, such as X server events and network messages.
- Files and devices in known locations (such as a data file name passed to or computed by the program).

Programs can emit results in all the same ways (with output going to standard output).

Some Unix programs are graphical, some have screen-oriented character interfaces, and some use a starkly simple text-filter design unchanged from the days of mechanical teletypes. To the uninitiated, it is often far from obvious why any given program uses the style it does — or, indeed, why Unix supports such a plethora of interface styles at all.

Unix has several competing interface styles. All are still alive for a reason; they're optimized for different situations. By understanding the fit between task and interface style, you will learn how to choose the right styles for the jobs you need to do.

Applying the Rule of Least Surprise

The Rule of Least Surprise is a general principle in the design of all kinds of interfaces, not just software: "Do the least surprising thing". It's a consequence of the fact that human beings can only pay attention to one thing at one time (see *The Humane Interface* [Raskin]). Surprises in the interface focus that single locus of attention on the interface, rather than on the task where it belongs.

Thus, to design usable interfaces, it's best when possible not to design an entire new interface model. Novelty is a barrier to entry; it puts a learning burden on the user, so minimize it. Instead, think carefully about the experience and knowledge of your user base. Try to find functional similarities between your program and programs they are likely to already know about. Then mimic the relevant parts of the existing interfaces.

The Rule of Least Surprise should not be interpreted as a call for mechanical conservatism in design. Novelty raises the cost of a user's first few interactions with an interface, but poor design will make the interface needlessly painful forever. As in other sorts of design, rules are not a substitute for good taste and engineering judgment. Consider your tradeoffs carefully — and consider them from the *user's* point of view. The bias implied by the Rule of Least Surprise is a good one to hold consciously, mainly because interface designers (like other programmers) have an unconscious tendency to be too clever for the user's good.

One implication of the Rule of Least Surprise is this: Wherever possible, allow the user to delegate interface functions to a familiar program. We already observed in Chapter 7 that, if your program requires the user to edit significant amounts of text, you should write it to call an editor (specifiable by the user) rather than building in your own integrated editor. This will enable the *users*, who know their preferences better than you, to choose the least surprising alternative.

Elsewhere in this book we have advocated symbiosis and delegation as tactics for promoting code reuse and minimizing complexity. The point here is that when users can intercept the delegation, and direct it to an agent of their own choice, these techniques become not merely economical for the developer but actively empowering to users.

Further: When you can't delegate, emulate. The purpose of the Rule of Least Surprise is to reduce the amount of complexity a user must absorb to use an interface. Continuing the editor example, this means that if you must implement an embedded editor, it's best if the editor commands are a subset of those for a well-known general-purpose editor. (Or more than one. Both *bash* and *ksh* have command-line editors that allow the user to choose between *vi* and *Emacs* editing styles.)

Under the Unix versions of the Netscape and Mozilla Web browsers, for example, fill-in fields in forms recognize a subset of the default bindings for the *Emacs* editor. Control-A goes to start of line, Control-D deletes the next character, and so forth. This choice helps people who know *Emacs*, and leaves others no worse off than an arbitrary, idiosyncratic command set would have. The only way it could have been bettered was by choosing key bindings associated with some editor significantly more widely used than *Emacs*; and among Netscape's original user population there was no such animal.

These principles can be applied in many other areas of interface design. They suggest, for example, that it is deeply foolish to create novel document formats for an on-line help system when users are comfortable with an HTML Web browser. Or even that if you are designing an arcade-style game, it is wise to look at the gesture sets of previous games to see if you can give new users a feeling of comfort by allowing them to transfer joystick skills learned in other games.

History of Interface Design on Unix

Unix predates the modern graphics-intensive style of software interface design. For over a decade after the first Unix in 1969, command-line interfaces (CLIs) on teletypes and dumb text-mode terminals were the norm. Most of the basic Unix toolset (programs like ls(1), cat(1), and grep(1)) still reflect this heritage.

Gradually, after 1980, Unix evolved support for screen-painting on character-cell terminals. Programs began to mix command-line and visual interfaces, with common commands often bound to keystrokes that would not be echoed to the screen. Some of the early programs written in this style (often called 'curses' programs, after the screen-painting cursor-control library normally used to implement them, or 'roguelike' after the first application to use curses) are still used today; notable examples include the dungeon-crawling game rogue(1), the vi(1) text editor, and (from a few years later) the elm(1) mailer and its modern descendant mutt(1).

A few years later in the mid-1980s, the computing world as a whole began to assimilate the results of the pioneering work on graphical user interfaces (GUIs) that had been going on at Xerox's Palo Alto Research Center since the early 1970s. On personal computers, the Xerox PARC work inspired the Apple Macintosh interface and through that the design of Microsoft Windows. Unix's adaptation of these ideas took a rather more complicated path.

Around 1987 the X windowing system outcompeted several early contenders and prototype efforts to become the standard graphical-interface facility for Unix. Whether this was a good or a bad thing has remained a topic of debate ever since; some of the other contenders (notably Sun's Network Window System or NeWS) were arguably rather more powerful and elegant. X, however, had one overriding virtue; it was open source. The code had been developed at MIT by a research group more interested in exploring the problem space than in creating a product, and it remained freely redistributable and modifiable. It was thus able to attract support from a wide range of developers and sponsoring corporations who would have been reluctant to line up behind a single vendor's closed product. (This, of course, prefigured an important theme in the breakout of the Linux operating system ten years later.)

The designers of X decided early on that X would support "mechanism, not policy". Their objective was to make X as flexible and portable across platforms as possible, while putting as few constraints on the look and feel of X programs as they could manage. Look and feel, they decided, would be handled by 'toolkits' — libraries calling X services linked to user programs. X would also be designed to support multiple window managers,[106] and would not require a window manager to have any special privileges or uniquely close integration with X's machinery.

This approach was the polar opposite of that taken by the Macintosh and Windows commercial products, which enforced particular look-and-feel policies by designing them right into the system. The difference in approach ensured that X would have a long-run evolutionary advantage by remaining adaptable as new discoveries were made about the human factors in interface design

[106] A window manager handles associations between windows on the screen and running tasks. Window managers handle behaviors like title bars, placement, minimizing, maximizing, moving, resizing, and shading windows.

— but it also ensured that the X world would be divided by multiple toolkits, a profusion of window managers, and many experiments in look and feel.

Since the mid-1990s X has become ubiquitous even on the lowest-end personal Unix machines. Use of Unix from text-mode terminals, as opposed to graphics-capable computer consoles, has sharply declined and seems headed for extinction. Accordingly, the use of curses-style interfaces for new applications is also in decline; most new applications that would formerly have been designed in that style now use an X toolkit. It is instructive to note that Unix's older CLI design tradition is still quite vigorous and successfully competes with X in many areas.

It is also instructive to note that there are a few specific application areas in which curses-style (or 'roguelike') character-cell interfaces remain the norm — especially text editors and interactive communications programs such as mailers, newsreaders, and chat clients.

For historical reasons, then, there is a wide range of interface styles in Unix programs. Line-oriented, character-cell screen-oriented, and X-based — with the X-based world somewhat balkanized by the competition between multiple X toolkits and window managers (though this is less an issue in 2003 than was the case five or even three years ago).

Evaluating Interface Designs

All these interface styles survive because they are adapted for different jobs. When making design decisions about a project, it's important to know how to pick a style (or combine styles) that will be appropriate to your application and your user population.

We will use five basic metrics to categorize interface styles: *concision*, *expressiveness*, *ease*, *transparency*, and *scriptability*. We've already used some of these terms earlier in this book in ways that were preparation for defining them here. They are comparatives, not absolutes; they have to be evaluated with respect to a particular problem domain and with some knowledge of the users' skill base. Nevertheless, they will help organize our thinking in useful ways.

A program interface is 'concise' when the length and complexity of actions required to do a transaction with it has a low upper bound (the measurement might be in keystrokes, gestures, or seconds of attention required). Concise interfaces pack a lot of leverage into a relatively few bits or state changes.

Interfaces are ‘expressive’ when they can readily be used to command a wide variety of actions. The *most* expressive interfaces can command combinations of actions not anticipated by the designer of the program, but which nevertheless give the user useful and consistent results.

The difference between concision and expressiveness is an important one. Consider two different ways of entering text: from a keyboard, or by picking characters from a screen display with mouse clicks. These have equal expressiveness, but the keyboard is more concise (as we can easily verify by comparing average text-entry speeds). On the other hand, consider two dialects of the same programming language, one with a complex-number type and one not. Within the problem domain they have in common, their concision will be identical; but for a mathematician or electrical engineer, the dialect with complex numbers will be much more expressive.

The ‘ease’ of an interface is inversely proportional to the mnemonic load it puts on the user — how many things (commands, gestures, primitive concepts) the user has to remember specifically to support using that interface. Programming languages have a high mnemonic load and low ease; menus and well-labeled on-screen buttons are simpler.

Recall that we devoted an entire earlier chapter to ‘transparency’. In that chapter we touched on the idea of interface transparency, and gave the *audacity audio editor* as one superb example of it. But we were then much more interested in transparency of a different kind, one that relates to the structure of code rather than of user interfaces. We therefore described UI transparency in terms of its effect (nothing obtrudes between the user and the problem domain) rather than the specific features of design that produce it. Now it’s time to zero in on these.

The ‘transparency’ of an interface is how few things the user has to remember about the state of his problem, his data, or his program while *using* the interface. An interface has high transparency when it naturally presents intermediate results, useful feedback, and error notifications on the effects of a user’s actions. So-called WYSIWYG (What You See Is What You Get) interfaces are intended to maximize transparency, but sometimes backfire — especially by presenting an over-simplified view of the domain.

The related concept of discoverability applies to interface design, as well. A discoverable interface provides the user with assistance in learning it, such as a greeting message pointing to context-sensitive help, or explanatory balloon popups. Though discoverability has to be implemented in rather different ways for each of the interface styles we shall consider, the degree to which it is achievable is largely independent of interface style. Thus, we shall not use it as a metric in this discussion.

Note that transparency of code and design does not automatically imply transparency of interface, or vice versa! It is all too easy to point to code that has one but not the other.

The ‘scriptability’ of an interface is the ease with which it can be manipulated by other programs (e.g., through the IPC mechanisms discussed in Chapter 7). Scriptable programs are readily usable as components by other programs, reducing the need for costly custom coding and making it relatively easy to automate repetitive tasks.

That last point — automating repetitive tasks — deserves more attention than it usually gets. Unix programmers, administrators, and users develop a habit of thinking through the routine procedures they use, then packaging them so they no longer have to manually execute or even think about them any more. This habit depends on scriptable interfaces. It is a quiet but tremendous productivity booster not available in most other software environments.

It will be useful to bear in mind that humans and computer programs have very different cost functions with respect to these metrics. So do novice and expert human users in a particular problem domain. We’ll explore how the tradeoffs between them change for different user populations.

Tradeoffs between CLI and Visual Interfaces

The CLI style of early Unix has retained its utility long after the demise of teletypes for two reasons. One is that command-line and command-language interfaces are more expressive than visual interfaces, especially for complex tasks. The other is that CLI interfaces are highly scriptable — they readily support the combining of programs, as we discussed in detail in Chapter 7. Usually (though not always) CLIs have an advantage in concision as well.

The disadvantage of the CLI style, of course, is that it almost always has high mnemonic load (low ease), and usually has low transparency. Most people (especially nontechnical end users) find such interfaces relatively cryptic and difficult to learn.

> On the other hand, the ‘user-friendly’ GUIs of other operating systems have their own problems. Finding the right buttons to push is like playing Adventure: the interfaces are just as burdensome as any Unix command line interface, save that one can in theory find the treasure by sufficient exploration. In Unix, one needs the manual.
>
> — <author>BrianKernighan</author>

Database queries are a good example of the kind of interface for which pushing buttons is not just burdensome but extremely limiting. Neither keystroke commands to a full-screen character interface nor GUI gestures on a graphic display can express typical actions in the problem domain as expressively or concisely as typing SQL direct to a server. And it is certainly easier to make a client program utter SQL queries than it would be to have it simulate a user clicking a GUI!

On the other hand, many non-technical database users are so resistant to having to remember SQL syntax that they prefer a less concise and less expressive full-screen or GUI interface.

SQL is a good example for illustrating another point. The most powerful CLIs are not ad-hoc collections of commands, but imperative minilanguages designed along the lines we described in Chapter 8. These minilanguages are the highest-power, highest-complexity end of the CLI spectrum; they maximize expressiveness, but minimize ease. They are difficult to use and generally need to be discreetly veiled from ordinary end-users, but unbeatable when the capability and flexibility of the interface is the most important thing. When properly designed, they also score high on scriptability.

Some applications, unlike database queries, are naturally visual. Paint programs, Web browsers, and presentation software make three excellent examples. What these application domains have in common is that (a) transparency is extremely valuable, and (b) the primitive actions in the problem domain are themselves visual: “draw this”, “show me what I’m pointing at”, “put this here”.

The flip side of paint programs is that it is difficult to capture relationships within the pictures they are manipulating. It takes careful, thoughtful design to give the user any handle on the structure of images with repeated elements, for example. This is a general design problem with visual interfaces.

In Chapter 6 we looked at the Audacity sound file editor. Its interface design succeeds because it does a particularly clean job of mapping its audio application domain onto a simple set of visual representations (borrowed from equalizer displays on stereos). It does this by thoroughly following through the consequences of a single translation: sounds to waveform images. The visual operations are not a mere grab-bag of low-level tweaks; they are all tied to that translation.

In applications that are *not* naturally visual, however, visual interfaces are most appropriate for simple one-shot or infrequent tasks performed by novice users (a point the database example illustrates).

Resistance to CLI interfaces tends to decrease as users become more expert. In many problem domains, users (especially *frequent* users) reach a crossover point at which the concision and expressiveness of CLI becomes more valuable than avoiding its mnemonic load. Thus, for example,

computing novices prefer the ease of GUI desktops, but experienced users often gradually discover that they prefer typing commands to a shell.

CLIs also tend to gain utility as problems scale up and involve more in the way of canned, procedural and repetitive actions. Thus, for example, a WYSIWYG desktop-publishing program is usually the easiest route to composing relatively small and unstructured documents such as business letters. But for complex book-sized documents that are assembled from sections and may require many global format changes or structural manipulation during composition, a minilanguage formatter such as *troff*, *Tex*, or some XML-markup processor is usually a more effective choice (see Chapter 18 for more discussion of this tradeoff).

Even in domains that are naturally visual, scaling up the problem size tends to tilt the tradeoff toward a CLI. If you need to fetch and save one Web page from a given URL, point and click (or type and click) is fine. But for Web forms, you're going to use a keyboard. And if you need to fetch and save the pages corresponding to a given list of fifty URLs, a CLI client that can read URLs from standard input or the command line can save you a lot of unnecessary motion.

As another example, consider modifying the color table in a graphic image. If you want to change one color (say, to lighten it by an amount you will only know is right when you see it) a visual dialogue with a color-picker widget is almost mandatory. But suppose you need to replace the entire table with a set of specified RGB values, or to create and index large numbers of thumbnails. These are operations that GUIs usually lack the expressive power to specify. Even when they do, invoking a properly designed CLI or filter program will do the job far more concisely.

Finally (as we observed earlier on) CLIs are important in facilitating using programs from other programs. A GUI graphics editor that *can* handle making a batch of thumbnails for a list of files probably does it with a plugin written in a scripting language, calling an internal CLI of the graphics editor (as in the GIMP's script-fu facility). Unix environments bring the value of CLIs into sharper relief precisely because their IPC facilities are rich, have low overhead, and are easily accessible from user programs.

The explosion of interest in GUIs since 1984 has had the unfortunate effect of obscuring the virtues of CLIs. The design of consumer software, in particular, has become heavily skewed toward GUIs. While this is a good choice for the novice and casual users that constitute most of the consumer market, it also exacts hidden costs on more expert users as they run up against the expressiveness limits of GUIs — costs which steadily increase as the users take on more demanding problems. Most of these costs derive from the fact that GUIs are simply not scriptable at all — *every* interaction with them has to be human-driven.

Gentner & Nielsen sum up the tradeoff very well in *The Anti-Mac Interface* [Gentner-Nielsen]: “[Visual interfaces] work well for simple actions with a small number of objects, but as the number of actions or objects increases, direct manipulation quickly becomes repetitive drudgery. The dark side of a direct manipulation interface is that you have to manipulate everything. Instead of an executive who gives high-level instructions, the user is reduced to an assembly-line worker who must carry out the same task over and over”. Noted science-fiction writer Neal Stephenson made the same point, less directly but more entertainingly, in his brilliant and discursive essay *In the Beginning Was the Command Line* [Stephenson].

A typical Unix old hand’s take on this problem is rather less theoretical:

> The commercial world generally goes for the novice mode because (a) purchase decisions are often made on the basis of 30 seconds trial, and (b) it minimizes the demands on customer support to have only a dumbed-down GUI. I find many non-Unix systems very frustrating because, for example, they will provide no way to do something on a hundred or a thousand files; I want to write a script, and there’s no support for it. The basic problem is that they’ve assumed all users are novices all the time, and then they bash Unix because it doesn’t cater to that model.
>
> —
>
> <author>MikeLesk</author>

For the long haul, then — for serving both casual and expert users, for cooperating with other computer programs, and whether the problem domain is naturally visual or not — support for *both* CLI and visual interfaces is important. Unix’s history positions it well to meet both sets of needs. After presenting an indicative case study, we will examine the characteristic design patterns that the Unix tradition has evolved to meet them.

Case Study: Two Ways to Write a Calculator Program

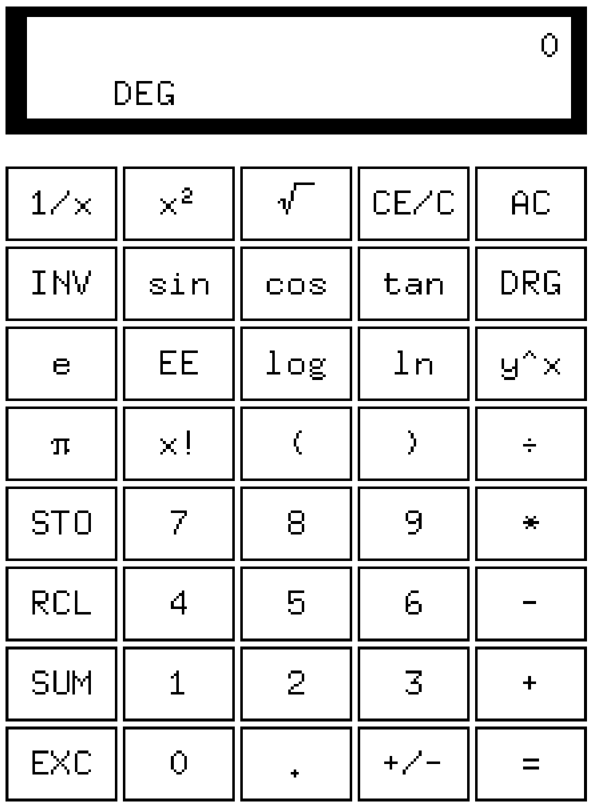
To be more concrete, let us contrast how the GUI and CLI styles can be usefully applied to the design of a simple interactive program: a desk calculator. Our examples for contrast are dc(1)/bc(1) and xcalc(1).

The original Unix desk calculator program, first distributed with Version 7, was dc(1)—a reverse-Polish-notation calculator that could handle unlimited-precision arithmetic. Later, an algebraic (infix notation) calculator language, bc(1), was implemented on top of dc (we used the relationship between these programs as a case study in Chapter 7, and again in Chapter 8). Both of these

programs use a CLI. You type an expression on standard input, you press enter, and the value of the expression is printed on standard output.

The xcalc(1) program, on the other hand, visually simulates a simple calculator, with clickable buttons and a calculator-style display.

Figure 11.1. The xcalc GUI.



The xcalc(1) approach is simpler to describe because it mimics an interface with which novice users will be familiar; the man page says, in fact, "The numbered keys, the +/- key, and the +, -, *, /, and = keys all do exactly what you would expect them to". All the capabilities of the program are

conveyed by the visible button labels. This is the Rule of Least Surprise in its strongest form, and a real advantage for infrequent and novice users who will never have to read a man page to use the program.

However, *xcalc* also inherits the almost complete non-transparency of a calculator; when evaluating a complex expression, you don't get to see and sanity-check your keystrokes — which can be a problem if, say, you misplace a decimal point in an expression like (2.51 + 4.6) * 0.3. There's no history, so you can't check. You'll get a result, but it won't be the result of the calculation you intended.

With the dc(1) and bc(1) programs, on the other hand, you can edit mistakes out of the expression as you build it. Their interface is more transparent, because you can see the calculation that is being performed at every stage. It is more expressive because the *dc*/*bc* interpreter, not being limited to what fits on a reasonably-sized visual mockup of a calculator, can include a much larger repertoire of functions (and facilities such as if/then/else, stored variables, and iteration). It also incurs, of course, a higher mnemonic load.

Concision is more of a toss-up; good typists will find the CLI more concise, while poor ones may find it faster to point and click. Scriptability is not a toss-up; *dc*/*bc* can easily be used as a filter, but xcalc can't be scripted at all.

The tradeoff between ease for novices and utility for expert users is very clear here. For casual use in situations where a mental-arithmetic error check is not hard, *xcalc* wins. For more complex calculations where the steps must not only be correct but must be *seen* to be correct, or in which they are most conveniently generated by another program, *dc*/*bc* wins.

Transparency, Expressiveness, and Configurability

Unix programmers inherit a strong bias toward making interfaces expressive and configurable. Like programmers from other traditions, they think about how to match their interfaces to the target audience — but they differ in how they deal with uncertainty about that target audience. Software developers whose experience is primarily with client operating systems default toward making interfaces simple; they are willing to sacrifice expressiveness to gain ease. Unix programmers default toward making interfaces expressive and transparent, and are more willing to sacrifice ease to get these qualities.

The results of this attitude have often been described as interfaces written "by programmers, for programmers". But this oversimplifies the matter in an important way. When a Unix programmer

opts for configurability and expressiveness over ease, he is not necessarily thinking of his audience as consisting solely of other programmers; rather, he is often acting on a gut-level instinct that in the absence of knowledge about end-users' intentions it is best not to patronize or second-guess them.

> The downside of this attitude (which is a close cousin to "mechanism, not policy") is a tendency to assume that when the highly configurable and expressive interface is done, the job is finished... even if the result is almost impossible for anyone else to use without lengthy study. The flip side of configurability is an urgent need for good defaults and an easy way to set everything to the default. The flip side of expressivity is a need for guidance — be it in the program or the documentation — on where to get started and how to achieve the most commonly-desired results.
>
> —
>
> <author>HenrySpencer</author>

The Rule of Transparency also has an influence. When a Unix programmer is writing to meet an RFC or other standard that defines a set of control options, he tends to assume that his job is to provide a complete and transparent interface to all of those options; whether or not he thinks any given one will actually be used is secondary. His job is mechanism; policy belongs to the user.

This mindset leads to a much stricter attitude about what constitutes standards conformance, one in which incomplete support is much less tolerable. In cases where a Macintosh or Windows developer would say "We don't need to support that feature of the standard; most users won't care, and it's too complicated for them", a Unix developer is likely to say "We don't know that nobody will ever want this feature or option, therefore we must support it".

These attitudes can lead to clashes when a Unix programmer is working with others, who are likely to interpret his design choices as a blithe willingness to burden users with technical details that are obscure, pointless, and even frightening. Mac or Windows programmers fear scaring away the many to serve the advanced needs of the few.

The Unix programmer, on the other hand, is likely to see defaulting away from expressiveness as a sort of cop-out or even betrayal of future users, who will know their own requirements better than the present implementer. Ironically, though the Unix attitude is often construed as a sort of programmer arrogance, it is actually a form of humility — one often acquired along with years of battle scars.

The extent to which the Unix attitude is appropriate varies. Whichever side of this divide you the reader are on, it is wise to learn to listen to the other, and wise to understand the premises behind the opposing point of view. Rather than falling into the trap of either intimidating users

or condescending to them, it may be possible to build transparent interfaces in which the advanced features are present but inconspicuous. The *audacity* and *kmail* case studies in Chapter 6 are good examples to follow.

Finally, a note about user-interface design for nontechnical end-users. This is a demanding art, and Unix programmers don't have a tradition of being very good at it. But with the ideas we've developed from examining the Unix tradition, it is possible to make one strong and useful statement about it. That is: when people say a user interface is *intuitive*, what they mean is that it (a) is discoverable, (b) is transparent in use, and (c) obeys the Rule of Least Surprise.[107] Of these three rules, Least Surprise is the least binding; initial surprises can be coped with if discoverability and transparency make longer-term use rewarding.

The user interfaces of today's cellphones (for example) have relatively high mnemonic load in that you have to maintain at least a rough mental map of the interface menus to use them rapidly without constantly having to spend attention on checking where you are in the hierarchy. But the better-designed ones rapidly become 'intuitive' for their users anyway, because they have these three qualities.

Intuitiveness is not quite the same quality as ease, because (as the cellphone example shows) people can develop what they think of as 'intuitions' about transparent interfaces that have fairly high mnemonic load, as long as simple operations are easy and there is a discovery path that allows them to assimilate the interface's more difficult corners one step at a time.

Unix Interface Design Patterns

In the Unix tradition, the tradeoffs we described above are met by well-established interface design patterns. Here is a bestiary of these patterns, with analyses and examples. We'll follow it with a discussion of how to apply them.

Note that this bestiary does not include GUI design patterns (though it includes a design pattern that can use a GUI as a component). There are no design patterns in graphical user interfaces themselves that are specifically native to Unix. A promising beginning of a discussion of GUI design patterns in general can be found at *Experiences — A Pattern Language for User Interface Design* [Coram-Lee].

[107] This insight comes to us from a nontechnical end-user who just happens to be the author's wife Catherine Raymond.

Also note that programs may have modes that fit more than one interface pattern. A program that has a compiler-like interface, for example, may behave as a filter when no file arguments are specified on the command line (many format converters behave like this).

The Filter Pattern

The interface-design pattern most classically associated with Unix is the *filter*. A filter program takes data on standard input, transforms it in some fashion, and sends the result to standard output. Filters are not interactive; they may query their startup environment, and are typically controlled by command-line options, but they do not require feedback or commands from the user in their input stream.

Two classic examples of filters are tr(1) and grep(1). The tr(1) program is a utility that translates data on standard input to results on standard output using a translation specification given on the command line. The grep(1) program selects lines from standard input according to a match expression specified on the command line; the resulting selected lines go to standard output. A third is the sort(1) utility, which sorts lines in input according to criteria specified on the command line and issues the sorted result to standard output.

Both grep(1) and sort(1) (but not tr(1)) can alternatively take data input from a file (or files) named on the command line, in which case they do not read standard input but act instead as though that input were the catenation of the named files read in the order they appear. (In this case it is also expected that specifying "-" as a filename on the command line will direct the program explicitly to read from standard input.) The archetype of such 'catlike' filters is cat(1), and filters are expected to behave this way unless there are application-specific reasons to treat files named on the command line differently.

When designing filters, it is well to bear in mind some additional rules, partly developed in Chapter 1:

1. *Remember Postel's Prescription: Be generous in what you accept, rigorous in what you emit.* That is, try to accept as loose and sloppy an input format as you can and emit as well-structured and tight an output format as you can. Doing the former reduces the odds that the filter will be brittle in the face of unexpected inputs, and break in someone's hand (or in the middle of someone's toolchain). Doing the latter increases the odds that your filter will someday be useful as an input to other programs.

2. *When filtering, never throw away information you don't need to.* This, too, increases the odds that your filter will someday be useful as an input to other programs. Information you discard is information that no later stage in a pipeline can use.

3. *When filtering, never add noise.* Avoid adding nonessential information, and avoid reformatting in ways that might make the output more difficult for downstream programs to parse. The most common offenders are cosmetic touches like headers, footers, blank/ruler lines, summaries and conversions like adding aligned columns, or writing a factor of "1.5" as "150%". Times and dates are a particular bother because they're hard for downstream programs to parse. Any such additions should be optional and controlled by switches. If your program emits dates, it's good practice to have a switch that can force them into ISO8601 YYYY-MM-DD and hh:mm:ss formats — or, better yet, use those by default.

The term "filter" for this pattern is long-established Unix jargon.

> "Filter" is indeed long-established. It came into use on day one of pipes. The term was a natural transferral from electrical-engineering usage: data flowed from source through filters to sink. Source or sink could be either process or file. The collective EE term, "circuit", was never considered, since the plumbing metaphor for data flow was already well established.
>
> —
>
> <author>DougMcIlroy</author>

Some programs have interface design patterns like the filter, but even simpler (and, importantly, even easier to script). They are cantrips, sources, and sinks.

The Cantrip Pattern

The cantrip interface design pattern is the simplest of all. No input, no output, just an invocation and a numeric exit status. A cantrip's behavior is controlled only by startup conditions. Programs don't get any more scriptable than this.

Thus, the cantrip design pattern is an excellent default when the program doesn't require any runtime interaction with the user other than fairly simple setup of initial conditions or control information.

Indeed, because scriptability is important, Unix designers learn to resist the temptation to write more interactive programs when cantrips will do. A collection of cantrips can always be driven from an interactive wrapper or shell program, but interactive programs are harder to script. Good style

therefore demands that you try to find a cantrip design for your tool before giving in to the temptation to write an interactive interface that will be harder to script. And when interactivity seems necessary, remember the characteristic Unix design pattern of separating the engine from the interface; often, the right thing is an interactive wrapper written in some scripting language that calls a cantrip to do the real work.

The console utility clear(1), which simply clears your screen, is the purest possible cantrip; it doesn't even take command-line options. Other classic simple examples are rm(1) and touch(1). The startx(1) program used to launch X is a complex example, typical of a whole class of daemon-summoning cantrips.

This interface design pattern, though fairly common, has not traditionally been named; the term "cantrip" is my invention. (In origin, it's a Scots-dialect word for a magic spell, which has been picked up by a popular fantasy-role-playing game to tag a spell that can be cast instantly, with minimal or no preparation.)

The Source Pattern

A *source* is a filter-like program that requires no input; its output is controlled only by startup conditions. The paradigmatic example would be ls(1), the Unix directory lister. Other classic examples include who(1) and ps(1).

Under Unix, report generators like ls(1), ps(1), and who(1) tend strongly to obey the source pattern, so their output can be filtered with standard tools.

The term 'source' is, as Doug McIlroy noted, very traditional. It is less common than it might be because 'source' has other important meanings.

The Sink Pattern

A *sink* is a filter-like program that consumes standard input but emits nothing to standard output. Again, its actions on the input data are controlled only by startup conditions.

This interface pattern is unusual, and there are few well-known examples. One is lpr(1), the Unix print spooler. It will queue text passed to it on standard input for printing. Like many sink programs, it will also process files named to it on the command line. Another example is mail(1) in its mail-sending mode.

Many programs that might appear at first glance to be sinks take control information as well as data on standard input and are actually instances of something like the *ed* pattern (see below).

The term *sponge* is sometimes applied specifically to sink programs like sort(1) that must read their entire input before they can process any of it.

The term ‘sink’ is traditional and common.

The Compiler Pattern

Compiler-like programs use neither standard output nor standard input; they may issue error messages to standard error, however. Instead, a compiler-like program takes file or resource names from the command line, transforms the names of those resources in some way, and emits output under the transformed names. Like cantrips, compiler-like programs do not require user interaction after startup time.

This pattern is so named because its paradigm is the C compiler, cc(1) (or, under Linux and many other modern Unixes, gcc(1)). But it is also widely used for programs that do (for example) graphics file conversions or compression/decompression.

A good example of the former is the gif2png(1) program used to convert GIF (Graphic Interchange Format) to PNG (Portable Network Graphics).[108] Good examples of the latter are the gzip(1) and gunzip(1) GNU compression utilities, almost certainly shipped with your Unix system.

In general, the compiler interface design pattern is a good model when your program often needs to operate on multiple named resources and can be written to have low interactivity (with its control information supplied at startup time). Compiler-like programs are readily scriptable.

The term “compiler-like interface” for this pattern is well-understood in the Unix community.

The *ed* pattern

All the previous patterns have very low interactivity; they use only control information passed in at startup time, and separate from the data. Many programs, of course, need to be driven by a continuing dialog with the user after startup time.

[108] Sources for this program, and other converters with similar interfaces, are available at the PNG website [http://www.cdrom.com/pub/png/].

In the Unix tradition, the simplest interactive design pattern is exemplified by ed(1), the Unix line editor. Other classic examples of this pattern include ftp(1) and sh(1), the Unix shell. The ed(1) program takes a filename argument; it modifies that file. On its input, it accepts command lines. Some of the commands result in output to standard output, which is intended to be seen immediately by the user as part of the dialog with the program.

An actual sample ed(1) session will be included in Chapter 13.

Many browserlike and editorlike programs under Unix obey this pattern, even when the named resource they edit is something other than a text file. Consider gdb(1), the GNU symbolic debugger, as an example.

Programs obeying the *ed* interface design pattern are not quite so scriptable as would be the simpler interface types resembling filters. You can feed them commands on standard input, but it is trickier to generate sequences of commands (and interpret any output they might ship back) than it is to just set environment variables and command-line options. If the action of the commands is not so predictable that they can be run blind (e.g., with a here-document as input and ignoring output), driving *ed*-like programs requires a protocol, and a corresponding state machine in the calling process. This raises the problems we noted in Chapter 7 during the discussion of slave process control.

Nevertheless, this is the simplest and most scriptable pattern that supports fully interactive programs. Accordingly, it is still quite useful as a component of the “separated engine and interface” pattern we’ll describe below.

The Roguelike Pattern

The roguelike pattern is so named because its first example was the dungeon-crawling game rogue(1) (see Figure 11.2) under BSD; the adjective “roguelike” for this pattern is widely recognized in Unix tradition. Roguelike programs are designed to be run on a system console, an X terminal emulator, or a video display terminal. They use the full screen and support a visual interface style, but with character-cell display rather than graphics and a mouse.

Figure 11.2. Screen shot of the original Rogue game.

```
                                 a) some food
                                 b) +1 ring mail [4] being worn
```

```
--------------------------           ########## c) a +1,+2 mace in hand
|                 +###############          d) a +1,+0 short bow
|                 |                    e) 28 +0,+0 arrows
-----------------+--------                    f) a short bow
            #                          i) a magnesium wand
            #                          g) a magnesium wand
          ###           ------------------ j) a potion of detect things
    ---------+----------     |             l) a scroll of teleportation
    |              |     #+          --press space to continue--
    |              |     #|          |           #
    |              +#######|           |         ##
    |              |      |          +##############
    ---------+----------     ---------------------          #
      ######                                      #
  -------+----------                                   ######
  |...........@..!|                                       #
  |...........%...|          -------------------            #
  |...............|         #+            |        #######
  |..............+#################|          |        #
  |...............|          |           +###########
  -------------------          ------------------
Level: 3  Gold: 73     Hp: 36(36)   Str: 14(16)  Arm: 4  Exp: 4/78
```

Commands are typically single keystrokes not echoed to the user (as opposed to the command lines of the *ed* pattern), though some will open a command window (often, though not always, the last line of the screen) on which more elaborate invocations can be typed. The command architecture often makes heavy use of the arrow keys to select screen locations or lines on which to operate.

Programs written in this pattern tend to model themselves on either vi(1) or emacs(1) and (obeying the Rule of Least Surprise) use their command sequences for common operations such as getting help or terminating the program. Thus, for example, one can expect one of the commands 'x', 'q', or 'C-x C-c' to terminate a program written to this pattern.

Some other interface tropes associated with this pattern include: (a) the use of one-item-per-line menus, with the currently-selected item indicated by bold or reverse-video highlighting, and (b) 'mode lines' — program status summaries carried on a highlighted screen line, often near the bottom or at the top of the screen.

The roguelike pattern evolved in a world of video display terminals; many of these didn't have arrow or function keys. In a world of graphics-capable personal computers, with character-cell terminals a fading memory, it's easy to forget what an influence this pattern exerted on design; but the early exemplars of the roguelike pattern were designed a few years before IBM standardized the PC keyboard in 1981. As a result, a traditional but now archaic part of the roguelike pattern is the use of the h, j, k, and l as cursor keys whenever they are not being interpreted as self-inserting characters in an edit window; invariably k is up, j is down, h is left, and l is right. This history also explains why older Unix programs tend not to use the ALT keys and to use function keys in a limited way if at all.

Programs obeying this pattern are legion: The vi(1) text editor in all its variants, and the emacs(1) editor; elm(1), pine(1), mutt(1), and most other Unix mail readers; tin(1), slrn(1), and other Usenet newsreaders; the lynx(1) Web browser; and many others. Most Unix programmers spend most of their time driving programs with interfaces like these.

The roguelike pattern is hard to script; indeed scripting it is seldom even attempted. Among other things, this pattern uses raw-mode character-by-character input, which is inconvenient for scripting. It's also quite hard to interpret the output programmatically, because it usually consists of sequences of incremental screen-painting actions.

Nor does this pattern have the visual slickness of a mouse-driven full GUI. While the point of using the full screen interface is to support simple kinds of direct-manipulation and menu interfaces, roguelike programs still require users to learn a command repertoire. Indeed, interfaces built on the roguelike pattern show a tendency to degenerate into a sort of cluttered wilderness of modes and meta-shift-cokebottle commands that only hard-core hackers can love. It would seem that this pattern has the worst of both worlds, being neither scriptable nor conforming to recent fashions in design for end-users.

But there must be some value in this pattern. Roguelike mailers, newsreaders, editors, and other programs remain extremely popular even among people who invariably run them through terminal emulators on an X display that supports GUI competitors. Moreover, the roguelike pattern is so pervasive that under Unix even GUI programs often emulate it, adding mouse and graphics support to a command and display interface that still looks rather roguelike. The X mode of emacs(1), and the xchat(1) client are good examples of such adaptation. What accounts for the pattern's continuing popularity?

Efficiency, and perceived efficiency, seem to be important factors. Roguelike programs tend to be fast and lightweight relative to their nearest GUI competitors. For startup and runtime speed,

running a roguelike program in an Xterm may be preferable to invoking a GUI that will chew up substantial resources setting up its displays and respond more slowly afterwards. Also, programs with a roguelike design pattern can be used over telnet links or low-speed dialup lines for which X is not an option.

Touch-typists often prefer roguelike programs because they can avoid taking their hands off the keyboard to move a mouse. Given a choice, touch-typists will prefer interfaces that minimize keystrokes far off the home row; this may account for a significant percentage of vi(1)'s popularity.

Perhaps more importantly, roguelike interfaces are predictable and sparing in their use of screen real estate on an X display; they do not clutter the display with multiple windows, frame widgets, dialog boxes, or other GUI impedimenta. This makes the pattern well suited for use in programs that must frequently share the user's attention with other programs (as is especially the case with editors, mailers, newsreaders, chat clients, and other communication programs).

Finally (and probably most importantly) the roguelike pattern tends to appeal more than GUIs to people who value the concision and expressiveness of a command set enough to tolerate the added mnemonic load. We saw above that there are good reasons for this preference to become more common as task complexity, use frequency, and user experience rise. The roguelike pattern meets this preference while also supporting GUI-like elements of direct manipulation as an *ed*-pattern program cannot. Thus, far from having only the worst of both worlds, the roguelike interface design pattern can capture some of the best.

The 'Separated Engine and Interface' Pattern

In Chapter 7 we argued against building monster single-process monoliths, and that it is often possible to lower the global complexity of programs by splitting them into communicating pieces. In the Unix world, this tactic is frequently applied by separating the 'engine' part of the program (core algorithms and logic specific to its application domain) from the 'interface' part (which accepts user commands, displays results, and may provide services such as interactive help or command history). In fact, this separated-engine-and-interface pattern is probably the one most characteristic interface design pattern of Unix.

(The other, more obvious candidate for that distinction would be filters. But filters are more often found in non-Unix environments than engine/interface pairs with bidirectional traffic between them. Simulating pipelines is easy; the more sophisticated IPC mechanisms required for engine/interface pairs are hard.)

Owen Taylor, maintainer of the GTK+ library widely used for writing user interfaces under X, beautifully brings out the engineering benefits of this kind of partitioning at the end of his note Why GTK_MODULES is not a security hole [http://www.gtk.org/setuid.html]; he finishes by writing "[T]he secure setuid program is a 500 line program that does only what it needs to, rather than a 500,000 line library whose essential task is user interfaces".

This is not a new idea. Xerox PARC's early research into graphical user interfaces led them to propose the "model-view-controller" pattern as an archetype for GUIs.

- The "model" is what in the Unix world is usually called an "engine". The model contains the domain-specific data structures and logic for your application. Database servers are archetypal examples of models.
- The "view" part is what renders your domain objects into a visible form. In a really well-separated model/view/controller application, the view component is notified of updates to the model and responds on its own, rather than being driven synchronously by the controller or by explicit requests for a refresh.
- The "controller" processes user requests and passes them as commands to the model.

In practice, the view and controller parts tend to be more closely bound together than either is to the model. Most GUIs, for example, combine view and controller behavior. They tend to be separated only when the application demands multiple views of the model.

Under Unix, application of the model/view/controller pattern is far more common than elsewhere precisely because there is a strong "do one thing well" tradition, and IPC methods are both easy and flexible.

An especially powerful form of this technique couples a policy interface (often a GUI combining view and controller functions) with an engine (model) that contains an interpreter for a domain-specific minilanguage. We examined this pattern in Chapter 8, focusing on minilanguage design; now it's time to look at the different ways that such engines can form components of larger systems of code.

There are several major variants of this pattern.

Configurator/Actor Pair

In a configurator/actor pair, the interface part controls the startup environment of a filter or daemon-like program which then runs without requiring user commands.

The programs fetchmail(1) and fetchmailconf(1) (which we've already used as case studies in discoverability and data-driven programming and will encounter again as language case studies in Chapter 14) are a good example of a configurator/actor pair. *fetchmailconf* is the interactive dotfile configurator that ships with fetchmail. *fetchmailconf* can also serve as a GUI wrapper that runs fetchmail in either foreground or background mode.

This design pattern enables both *fetchmail* and *fetchmailconf* to specialize in what they do well, and indeed to be written in different languages appropriate to their task domains. Fetchmail, which usually runs in background as a daemon, need not be bloated with GUI code. Conversely, *fetchmailconf* can specialize in elaborate GUIness without exacting size and complexity costs from fetchmail. Finally, because the information channels between them are narrow and well-defined, it remains possible to drive *fetchmail* from the command line and from scripts other than *fetchmailconf*.

The term "configurator/actor" is my invention.

Spooler/Daemon Pair

A slight variant of the configurator/actor pair can be useful in situations that require serialized access to a shared resource in a batch mode; that is, when a well-defined job stream or sequence of requests requires some shared resource, but no individual job requires user interaction.

In this spooler/daemon pattern, the spooler or front end simply drops job requests and data in a spool area. The job requests and data are simply files; the spool area is typically just a directory. The location of the directory and the format of the job requests are agreed on by the spooler and daemon.

The daemon runs forever in background, polling the spool directory, looking there for work to do. When it finds a job request, it tries to process the associated data. If it succeeds, the job request and data are deleted out of the spool area.

The classic example of this pattern is the Unix print spooler system, lpr(1)/lpd(1). The front end is lpr(1); it simply drops files to be printed in a spool area periodically scanned by *lpd*. *lpd*'s job is simply to serialize access to the printer devices.

Another classic example is the pair at(1)/atd(1), which schedules commands for execution at specified times. A third example, historically important though no longer in wide use, was UUCP

— the Unix-to-Unix Copy Program commonly used as a mail transport over dial-up lines before the Internet explosion of the early 1990s.

The spooler/daemon pattern remains important in mail-transport programs (which are batchy by nature). The front ends of mail transports such as sendmail(1) and qmail(1) usually make one try at delivering mail immediately, through SMTP over an outbound Internet connection. If that attempt fails, the mail will fall into a spool area; a daemon version or mode of the mail transport will retry the delivery later.

Typically, a spooler/daemon system has four parts: a job launcher, a queue lister, a job-cancellation utility, and a spooling daemon, In fact, the presence of the first three parts is a sure clue that there is a spooler daemon behind them somewhere.

The terms “spooler” and “daemon” are well-established Unix jargon. (‘Spooler’ actually dates back to early mainframe days.)

Driver/Engine Pair

In this pattern, unlike a configurator/actor or spooler/server pair, the interface part supplies commands to and interprets output from an engine after startup; the engine has a simpler interface pattern. The IPC method used is an implementation detail; the engine may be a slave process of the driver (in the sense we discussed in Chapter 7) or the engine and driver may communicate through sockets, or shared memory, or any other IPC method. The key points are (a) the interactivity of the pair, and (b) the ability of the engine to run standalone with its own interface.

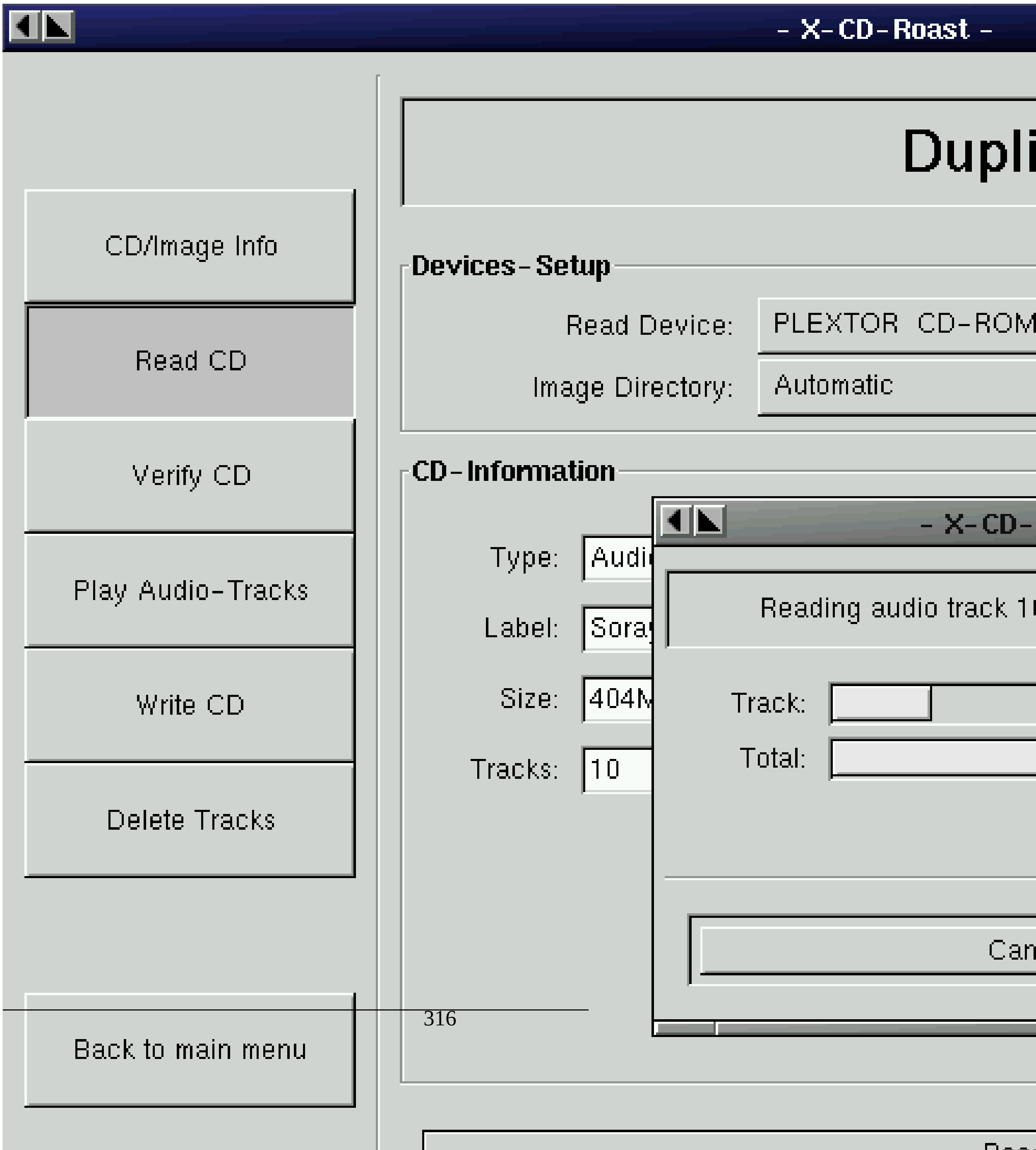
Such pairs are trickier to write than configurator/actor pairs because they are more tightly and intricately coupled; the driver must have knowledge not merely about the engine’s expected startup environment but about its command set and response formats as well.

When the engine has been designed for scriptability, however, it is not uncommon for the driver part to be written by someone other than the engine author, or for more than one driver to front-end a given engine. An excellent example of both is provided by the programs gv(1) and ghostview(1), which are drivers for gs(1), the Ghostscript interpreter. GhostScript renders PostScript to various graphics formats and lower-level printer-control languages. The gv and ghostview programs provide GUI wrappers for GhostScript’s rather idiosyncratic invocation switches and command syntax.

Another excellent example of this pattern is the *xcdroast*/*cdrtools* combination. The cdrtools distribution provides a program cdrecord(1) with a command-line interface. The *cdrecord* code

specializes in knowing everything about talking to CD-ROM hardware. *xcdroast* is a GUI; it specializes in providing a pleasant user experience. The xcdroast(1) program calls cdrecord(1) to do most of its work.

Figure 11.3. The Xcdroast GUI.



xcdroast also calls other CLI tools: cdda2wav(1) (a sound file converter) and mkisofs(1) (a tool for creating ISO-9660 CD-ROM file system images from a list of files). The details of how these tools are invoked are hidden from the user, who can think in terms centered on the task of making CDs rather than having to know directly about the arcana of sound-file conversion or file-system structure. Equally important, the implementers of each of these tools can concentrate on their domain-specific expertise without having to be user-interface experts.

> A key pitfall of driver/engine organization is that frequently the driver must understand the state of the engine in order to reflect it to the user. If the engine action is practically instantaneous, it's not a problem, but if the engine can take a long time (e.g., when accessing many URLs) the lack of feedback can be a significant issue. A similar problem is responding to errors. For example, the traditional (although not very Unix-like) confirmation question about whether it's OK to overwrite a file that already exists is kind of painful to write in the driver/engine world; the engine, which detects the problem, has to ask the driver to do the confirmation prompting.
>
> —
> <author>SteveJohnson</author>

It's important to design the engine so that it not only does the right thing, but also notifies the driver about what it's doing so the driver can present a graceful interface with appropriate feedback.

The terms "driver" and "engine" are uncommon but established in the Unix community.

Client/Server Pair

A client/server pair is like a driver/engine pair, except that the engine part is a daemon running in background which is not expected to be run interactively, and does not have its own user interface. Usually, the daemon is designed to mediate access to some sort of shared resource — a database, or a transaction stream, or specialized shared hardware such as a sound device. Another reason for such a daemon may be to avoid performing expensive startup actions each time the program is invoked.

Yesterday's paradigmatic example was the ftp(1)/ftpd(1) pair that implements FTP, the File Transfer Protocol; or perhaps two instances of sendmail(1), sender in foreground and listener in background, passing Internet email. Today's would have to be any browser/web server pair.

However, this pattern is not limited to communication programs; another important case is in databases, such as the psql(1)/postmaster(1) pair. In this one, *psql* serializes access to a shared

database managed by the postgres daemon, passing it SQL requests and presenting data sent back as responses.

These examples illustrate an important property of such pairs, which is that the cleanliness of the protocol that serializes communication between them is all-important. If it is well-defined and described by an open standard, it can become a tremendous opportunity for leverage by insulating client programs from the details of how the server's resource is managed, and allowing clients and servers to evolve semi-independently. All separated-engine-and-interface programs potentially get this kind of benefit from clean separation of function, but in the client/server case the payoffs for getting it right tend to be particularly high exactly because managing shared resources is intrinsically difficult.

Message queues and pairs of named pipes can be and have been used for front-end/back-end communication, but the benefits of being able to run the server on a different machine from the client are so great that nowadays almost all modern client-server pairs use TCP/IP sockets.

The CLI Server Pattern

It's normal in the Unix world for server processes to be invoked by harness programs[109] such as inetd(8) in such a way that the server sees commands on standard input and ships responses to standard output; the harness program then takes care of ensuring that the server's stdin and stdout are connected to a specified TCP/IP service port. One benefit of this division of labor is that the harness program can act as a single security gatekeeper for all of the servers it launches.

One of the classic interface patterns is therefore a CLI server. This is a program which, when invoked in a foreground mode, has a simple CLI interface reading from standard input and writing to standard output. When backgrounded, the server detects this and connects its standard input and standard output to a specified TCP/IP service port.

In some variants of this pattern, the server backgrounds itself by default, and has to be told with a command-line switch when it should stay in foreground. This is a detail; the essential point is that most of the code neither knows nor cares whether it is running in foreground or a TCP/IP harness.

POP, IMAP, SMTP, and HTTP servers normally obey this pattern. It can be combined with any of the server/client patterns described earlier in this chapter. An HTTP server can also act as a harness

[109] A harness program is a wrapper whose job it is to make some special sort of resource available to the program(s) it calls. The term is most often used for test harnesses, which make available test loads and (often) examples of correct output for the actual output to be checked against.

program; the CGI scripts that supply most live content on the Web run in a special environment provided by the server where they can take input (form arguments) from standard input, and write the generated HTML that is their result to standard output.

Though this pattern is quite traditional, the term “CLI server” is my invention.

Language-Based Interface Patterns

In Chapter 8 we examined domain-specific minilanguages as a means of pushing program specification up a level, gaining flexibility, and minimizing bugs. These virtues make the language-based CLI an important style of Unix interface — one exemplified by the Unix shell itself.

The strengths of this pattern are well illustrated by the case study earlier in the chapter comparing dc(1)/bc(1) with xcalc(1). The advantages that we observed earlier (the gain in expressiveness and scriptability) are typical of minilanguages; they generalize to other situations in which you routinely have to sequence complex operations in a specialized problem domain. Often, unlike the calculator case, minilanguages also have a clear advantage in concision.

One of the most potent Unix design patterns is the combination of a GUI front end with a CLI minilanguage back end. Well-designed examples of this type are necessarily rather complex, but often a great deal simpler and more flexible than the amount of ad-hoc code that would be necessary to cover even a fraction of what the minilanguage can do.

This general pattern is not, of course, unique to Unix. Modern database suites everywhere normally consist of one or more GUI front ends and report generators, all of which talk to a common back-end using a query language such as SQL. But this pattern mainly evolved under Unix and is still much better understood and more widely applied there than elsewhere.

When the front and back ends of a system fulfilling this design pattern are combined in a single program, that program is often said to have an ‘embedded scripting language’. In the Unix world, *Emacs* is one of the best-known exemplars of this pattern; refer to our discussion of it in Chapter 8 for some advantages.

The script-fu facility of GIMP is another good example. GIMP is a powerful open-source graphics editor. It has a GUI resembling that of Adobe Photoshop. Script-fu allows GIMP to be scripted using Scheme (a dialect of Lisp); scripting through Tcl, or Perl or Python is also available. Programs written in any of these languages can call GIMP internals through its plugin interface. The

demonstration application for this facility is a Web page[110] which allows people to construct simple logos and graphic buttons through a CGI interface that passes a generated Scheme program to an instance of GIMP, and returns a finished image.

Applying Unix Interface-Design Patterns

To facilitate scripting and pipelining (see Chapter 7) it is wise to choose the simplest interface pattern possible — that is, the pattern with the fewest channels to the environment and the least interactivity.

In many of the single-component patterns described above, it is emphasized that the pattern does not require user interaction after startup time. When the 'user' is often expected to be another program (and thus to lack the range and flexibility of a human brain) this is a very valuable feature, maximizing scriptability.

We've seen that different interface design patterns optimize for traits valuable in differing circumstances. In particular, there is a strong and inherent tension between the GUIs and design patterns appropriate for novice and nontechnical end-users (on the one hand) and those which serve expert users and maximize scriptability (on the other).

One way around this dilemma is to make programs with modes that exhibit more than one pattern. An excellent example is the Web browser lynx(1). It normally has a roguelike interface for interactive use, but can be called with a `-dump` option that makes it into a source, formatting a specified Web page to text dumped on standard output.

Such dual-mode interfaces, however, are not normally attempted when the program has to have a true GUI. The reasons for this are partly historical, but mostly have to do with controlling global complexity. GUIs tend to require complex startup configurations and large volumes of specialized code; these features coexist uneasily with the simpler patterns. In the worst case, a dual-mode GUI/non-GUI program could require two separate command-interpreter loops, with all that implies in the way of code bloat and potential inconsistencies.

Thus, when "choose the simplest pattern" conflicts with a requirement to produce a GUI, the Unix way is to split the program in two, applying the 'separated engine and interface' design pattern.

In fact, by combining a theme from Chapter 7 with this idea, we can perhaps name a new design pattern emerging under Linux and other modern, open-source Unixes where GUIs are not merely a reluctant add-on but an active focus of lots of development effort.

[110] Script-Fu page [http://www.xcf.berkeley.edu/~gimp/script-fu/script-fu.html].

The Polyvalent-Program Pattern

A polyvalent program has the following traits:

1. The program's application-domain logic lives in a library with a documented API, which can be linked to other programs. The program's interface logic to the rest of the world is a thin layer over the library. Or perhaps there are several layers with different UI styles, any of which the library can be linked to.

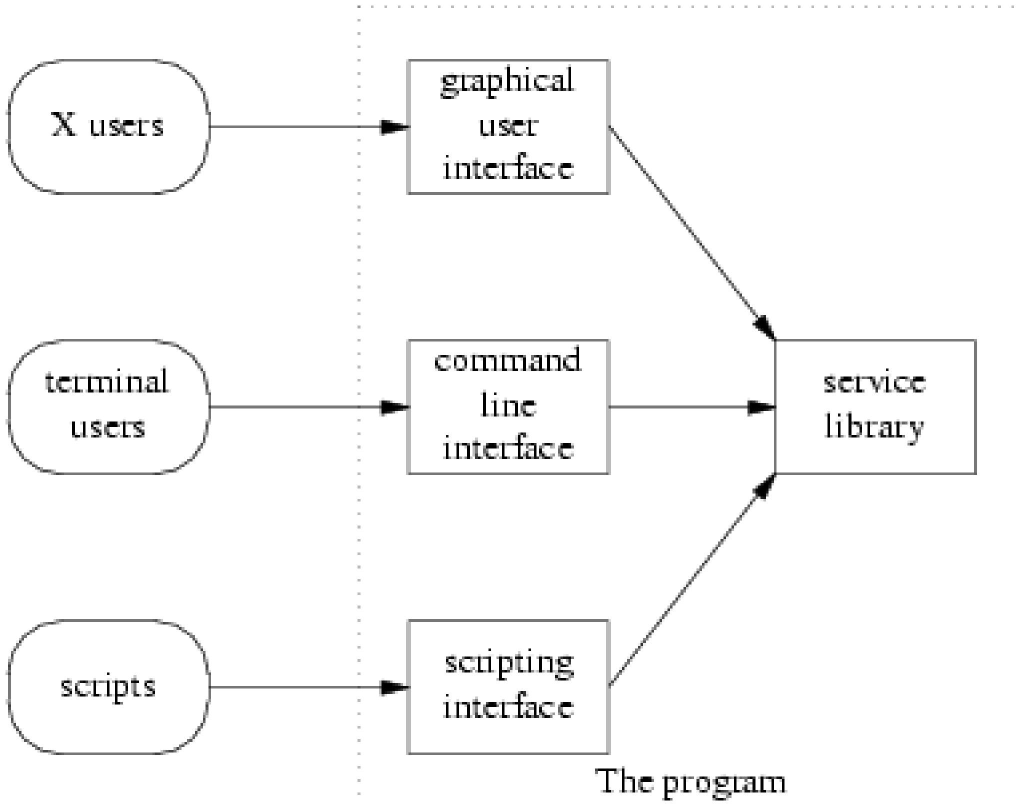
2. One UI mode is a cantrip, compiler-like or CLI pattern that executes its interactive commands in batch mode.

3. One UI mode is a GUI, either linked directly to the core library or acting as as a separate process driving the CLI interface.

4. One UI mode is a scripting interface using a modern general-purpose scripting language like Perl, Python, or Tcl.

5. Optional extra: One UI mode is a roguelike interface using curses(3).

Figure 11.4. Caller/callee relationships in a polyvalent program.



Notably, the GIMP actually fulfills this pattern.

The Web Browser as a Universal Front End

Separating your CLI back end from a GUI interface has become an even more attractive strategy since the transformation of computing by the World Wide Web in the mid-1990s. For a large class of applications, it makes increasing sense not to write a custom GUI front end at all, but rather to press Web browsers into service in that role.

This approach has many advantages. The most obvious is that you don't have to write procedural GUI code — instead, you can describe the GUI you want in languages (HTML and JavaScript) that are specialized for it. This avoids a lot of expensive and complex single-purpose coding and often more than halves the total project effort. Another is that it makes your application instantly Internet-ready; the front end may be on the same host as the back end, or may be a thousand miles away. Yet another is that all the minor presentation details of the application (such as fonts and color) are no longer your back end's problem, and indeed can be customized by users to their own tastes through mechanisms like browser preferences and cascading style sheets. Finally, the uniform elements of the Web interface substantially ease the user's learning task.

There are disadvantages. The two most important are (a) the batch style of interaction that the Web enforces, and (b) the difficulties of managing persistent sessions using a stateless protocol. Though these are not exclusively Unix issues, we'll discuss them here — because it's very important to think clearly on the *design* level about when it's worthwhile to accept or work around these constraints.

CGI, the Common Gateway Interface through which a browser can invoke a program on the server host, does not support fine-grained interactivity well. Nor do the templating systems, application servers, and embedded server scripts that are gradually replacing it (in a mild abuse of language, we will use CGI for all of these in this section).

You can't do character-by-character or GUI-gesture-by-GUI-gesture I/O through a CGI gateway; instead, you have to fill out an HTML form and click a submit button that sends the form contents to a CGI script. The CGI script then runs and the server hands you back a page of HTML that it generated (which may itself be another CGI form).

This is essentially a batch style of interaction, not that far removed in concept from dropping punched cards in an input hopper and getting back a printout. It can be made more palatable by using JavaScript to interact with the user, batching up transactions into messages to be shipped to the server.

Java applets can open up their own character-stream connections back to the server to support smoother interactivivity. But Java has technical problems (it can only use a fixed display area on the page, and can't change the portion of the display outside that rectangle) and much worse political ones (proprietary licensing from Sun has stalled Java deployment and made others reluctant to commit to it; you can't count on every user's browser to support it).

Both Java and JavaScript can run into browser incompatibilities, as well. Microsoft's resistance to implementing JDK 1.2 and Swing on Internet Explorer is a serious problem for Java applets, and

differing Javascript version levels can also break your application (though Javascript bugs are easier to fix). Nevertheless, it is frequently less effort to work around these problems than it would be to write and deploy a custom front end. A problem harder to work around is that a growing number of sophisticated users routinely disable Java and even JavaScript in their browsers because of security problems and interface abuses.

As an independent issue, it is tricky to maintain session information across multiple CGI forms. The server doesn't keep any state about client sessions between CGI transactions, so you can't rely on it to connect later form submissions with earlier ones by the same user. There are two standard dodges around this: chained forms and browser cookies.

When you chain forms, you arrange for the CGI for the first form to generate a unique ID in an invisible field of the second form, and for the second and all subsequent forms to pass that ID to their successors. Cookies give a similar effect in a less direct way analogous to environment variables (see any of the hundreds of books on CGI design for details). In either case, your CGI has to use the ID as a session index (or cookies to cache state directly) and to handle multiplexing the sessions explicitly.

It is often possible to live with these restrictions. Many nontrivial applications can fit into a single form and response, evading both problems. Even when this isn't true and the application requires multiple forms, the complexity and cost savings from not having to build and distribute a specialized front end are so large that they can easily pay for the effort required to write CGIs smart enough to do their own session tracking.

The session management problem can be addressed with application servers like Zope or Enhydra which provide a session abstraction, and services like user authentication to programs embedded inside them. The drawback of these programs is identical to their advantage: the fact that they make it easier to keep per-user state on the server. That per-user state can be a problem; it eats resources, and it has to be timed out, because between transactions there is no way to know that the user is still on the other end of the wire.

As usual, the best advice is to choose the simplest pattern possible. Resist the temptation to do a heavyweight design relying on Java or an application server when simple CGIs and cookies will do the job.

One problem with the browser-as-universal-front-end approach is that CGI back ends aren't readily separable from the browser environment, so it can be hard to script or automate transactions to the

back end. The Unix answer is a three-tier architecture — Web forms calling CGIs which call commands. The automation interface is the commands.

The way that browsers decouple front and back ends has larger implications. On the Web, locking in consumers to closed, proprietary protocols and APIs has become more difficult and less attractive as this trend has advanced. The economics of software development are therefore tilting toward HTML, XML, and other open, text-based Internet standards. This trend synergizes in interesting ways with the evolution of the open-source development model, which we'll survey in Chapter 19. In the world that the Web is creating, Unix's design tradition — including the approaches to interface design we've surveyed in this chapter — looks more at home than ever before.

Silence Is Golden

We cannot leave the subject of interactive user interfaces without exploring one of the oldest and most persistent design tropes of Unix, the Rule of Silence. We observed in Chapter 1 that well-designed Unix programs with nothing interesting or surprising to say should shut up, and suggested there are good reasons for this that have long outlasted the slow teletypes on which Unix was born.

Here's one: Programs that babble don't tend to play well with other programs. If your CLI program emits status messages to standard output, then programs that try to interpret that output will be put to the trouble of interpreting or discarding those messages (even if nothing went wrong!). Better to send only real errors to standard error and not to emit unrequested data at all.

Here's another: The user's vertical screen space is precious. Every line of junk your program emits is one less line of context still available on the user's display.

Here's a third: Junk messages are a careless waste of the human user's bandwidth. They're one more source of distracting motion on a screen display that may be mediating for more important foreground tasks, such as communication with other humans.

Go ahead and give your GUIs progress bars for long operations. That's good style — it helps the user time-share his brain efficiently by cuing him that he can go off and read mail or do other things while waiting for completion. But don't clutter GUI interfaces with confirmation popups except when you have to guard operations that might lose or trash data — and even then, hide them when the parent window is minimized, and bury them unless the parent window has focus.[111] Your job as an interface designer is to assist the user, not to gratuitously get in his face.

[111] If your windowing system supports translucent popups that intrude less between the user and the application, *use them*.

In general, it’s bad style to tell the user things he already knows ("Program <foo> is starting up...", or "Program <foo> is exiting" are two classic offenders). Your interface design as a whole should obey the Rule of Least Surprise, but the content of messages should obey a Rule of *Most* Surprise — be chatty only about things that deviate from what’s normally expected.

This rule has even greater force for confirmation prompts. Constantly asking for confirmation where the answer is almost always “yes” conditions the user to press “yes” without thinking about it, a habit that can have very unfortunate consequences. Programs should request confirmation only when there is good reason to suspect that the answer might be “no no no!” A confirmation request that is not a surprise is a strong hint of bad design. Any confirmation prompts at all may be a sign that what your interface really needs is an undo command.

If you want chatty progress messages for debugging purposes, disable them by default with a verbosity switch. Before releasing for production, relegate as many of the normal messages as possible to being displayed only when the verbosity switch is on.

Chapter 12. Optimization

Premature optimization is the root of all evil.
--
<author>C.A. R.Hoare</author>

This is going to be a very short chapter, because the main thing Unix experience teaches us about optimizing for performance is how to know when not to do it. A secondary lesson is that the most effective optimization tactics are usually things we do for other reasons, such as cleanness of design.

Don't Just Do Something, Stand There!

The most powerful optimization technique in any programmer's toolbox is to do nothing.

This very Zen advice is true for several reasons. One is the exponential effect of Moore's Law — the smartest, cheapest, and often *fastest* way to collect performance gains is to wait a few months for your target hardware to become more capable. Given the cost ratio between hardware and programmer time, there are almost always better things to do with your time than to optimize a working system.

We can get mathematically specific about this. It is almost never worth doing optimizations that reduce resource use by merely a constant factor; it's smarter to concentrate effort on cases in which you can reduce average-case running time or space use from $O(n^2)$ to $O(n)$ or $O(n \log n)$,[112] or similarly reduce from a higher order. Linear performance gains tend to be rapidly swamped by Moore's Law.[113]

[112] For readers unfamiliar with O notation, it is a way of indicating how the average running time of an algorithm changes with the size of its inputs. An O(1) algorithm runs in constant time. An $O(n)$ algorithm runs in a time that is predicted by `An + C`, where `A` is some unknown constant of proportionality and `C` is an unknown constant representing setup time. Linear search of a list for a specified value is $O(n)$. An $O(n^2)$ algorithm runs in time An^2 plus lower-order terms (which might be linear, or logarithmic, of any other function lower than a quadratic). Checking a list for duplicate values (by the naïve method, not sorting it) is $O(n^2)$. Similarly, $O(n^3)$ algorithms have an average run time predicted by the cube of problem size; these tend to be too slow for practical use. $O(\log n)$ is typical of tree searches. Intelligent choice of algorithm can often reduce running time from $O(n^2)$ to $O(\log n)$. Sometimes when we are interested in predicting an algorithm's memory utilization, we may notice that it varies as O(1) or $O(n)$ or $O(n^2)$; in general, algorithms with $O(n^2)$ or higher memory utilization are not practical either.

[113] The eighteen-month doubling time usually quoted for Moore's Law implies that you can collect a 26% performance gain just by buying new hardware in six months.

Another very constructive form of doing nothing is to not write code. The program can't be slowed down by code that isn't there. It can be slowed down by code that *is* there but less efficient than it could be — but that's a different matter.

Measure before Optimizing

When you have real-world evidence that your application is too slow, then (and *only* then) is the time to think about optimizing the code. But before you do more than think about optimizing, measure.

Recall Rob Pike's six rules in Chapter 1. One of the lessons that the original Unix programmers learned early is that intuition is a poor guide to where the bottlenecks are, even for one who knows the code in question intimately. Unixes, unlike most other operating systems, usually come with profilers; use them.

Reading profiler results is something of an art. There are a couple of recurring problems: one is instrumentation noise, another is the effect of imposed external latencies, and a third is overweighting of upper nodes in the call graph.

The instrumentation-noise problem is fundamental. Profilers work by inserting instructions that report execution time at the entry and exit points of subroutines, also at fixed intervals within the inline code of routines. These instructions themselves take time to execute. The effect is to reduce the dispersion of call times: very short subroutines tend to look more expensive than they are, with a lot of noise in their comparative call times, while for longer ones the instrumentation overhead is invisible.

Bearing instrumentation noise in mind, it's wise to assume that the times listed for the fastest, shortest subroutines are going to have a lot of froth and air in them. They can still be eating a lot of time if they are called very frequently, however, so pay particular attention to their call-count statistics.

The external-latency problem is also fundamental. There are various sorts of delay and distortion that can happen behind the profiler's back. The simplest is overhead from operations with unpredictable latency — disk and network accesses, cache fills, process-context switches, and the like. The problem is not so much that these overheads happen — they may actually be what you're trying to measure, especially if you're focusing on whole-system performance rather than just tuning a critical inner loop. The problem is that they have a random component that means the results from any individual profiling run may not be very useful.

One way to minimize the effects of these noise sources, and get a better picture of where the time is going in the average case, is to add together the results from a lot of profiling runs. There are a lot of good reasons to build test harnesses and test loads for your programs before you get to optimizing; the most important reason, usually far more important than performance tuning, is so you can regression-test your program for correctness as you change it. Once you've done this, being able to profile repeated tests under load is a nice side effect that will often give you better information than a few runs by hand.

Various effects tend to allocate time spent to calling routines rather than callees, overweighting upper modes in the call graph. Function-call overhead, for example, is often charged to the calling routine (whether or not this is true depends partly on your machine architecture and where the profiler is allowed to insert probes). Macros and inline functions, if your compiler supports them, won't show up in the profiling report at all; every bit of their time gets charged to the calling function.

More importantly, many time-reporting tools give a display in which time spent in subroutines is charged to the caller. (The gprof(1) profiler distributed with open-source Unixes has this trait.) Naïvely subtracting callee time from caller time won't give you a useful result if the same routine can have more than one caller — the effect would be to artificially deflate both callers' times. Especially nasty is the common case of a utility function with multiple call sites, some of which make lots of trivial calls and others of which make a few complicated ones.

To get more transparent results, factor your code so that upper-level routines consist as much as possible of calls to lower-level routines, rather than in-line code. If you keep the overhead of upper-level control logic to a minimum, the call structure of the code will tend to organize the profile report in a way that is relatively easy to read.

You'll get more insight from using profilers if you think of them less as ways to collect individual performance numbers, and more as ways to learn how performance varies as a function of interesting parameters (e.g., problem size, CPU speed, disc speed, memory size, compiler optimization, or whatever else is relevant). Try fitting those numbers to a model, using open-source software like R or a good-quality proprietary tool like MATLAB.

> The natural smoothing of the data that results from model fitting tends to focus on the big effects and cover up the small, noisy ones. For example, by fitting a cubic to the matrix inversion routine in MATLAB on random matrices from 10×10 to 1000×1000, it is clear that we actually have three cubics, with clearly defined boundaries, that correspond roughly to “in cache”, “in memory but out of cache”,

and “out of memory”. The data shows us this effect even if weren’t looking for it, just by looking at the deviations from the best fit.

—
<author>SteveJohnson</author>

Nonlocality Considered Harmful

The most effective way to optimize your code is to keep it small and simple. We’ve been through lots of good reasons to keep it small and simple earlier in this book. Here’s a new one: you want the central data structures and the time-critical loops in your code never to fall out of cache.

Consider your target machine as a hierarchy of memory types arranged by distance from the processor. There are the processor’s own registers; its instruction pipeline; the level-one (L1) cache; the level-two (L2) cache; possibly a level-three (L3) cache; main memory (what Unix old hands still quaintly call ‘core’); and the disk drives where swap space lives. Technologies like SMP, shared-memory clusters, and non-uniform memory access (NUMA) add more layers to the picture but only widen the overall spread.

Every kind of access to that stack is getting faster. Processor cycles are almost free, outside of a few demanding applications like modeling nuclear explosions or real-time video compression. But what’s also happening is that the speed ratios between layers in the storage hierarchy are all increasing as processor speeds go up. Thus, the relative cost of a cache miss is increasing.

So we have an interesting paradox. As machine resources plummet, the expected cost of large data structures falls — but because the cost spread between adjacent cache levels is also going up, the performance impact of being just large enough to break a cache boundary is also rising.

“Small is beautiful” is therefore better advice than ever, particularly with regard to central data structures that must live in the fastest possible cache. The advice applies to code as well; the average instruction spends more time being loaded than it does executing.

This turns some traditional advice on its head. Compiler optimizations like loop unrolling, which get rid of relatively expensive machine instructions in return for an increase in total code size, may no longer be worth doing. Another example is precomputing small tables — for example, a table of sin(x) by degree for optimizing rotations in a 3D graphics engine will take 365 × 4 bytes on a modern machine. Before processors got enough faster than memory to demand caching, this was

an obvious speed optimization. Nowadays it may be faster to recompute each time rather than pay for the percentage of additional cache misses caused by the table.

But in the future, this might turn around again as caches grow larger. More generally, many optimizations are temporary and can easily turn into pessimizations as cost ratios change. The only way to know is to measure and see.

Throughput vs. Latency

Another effect of fast processors is that performance is usually bounded by the cost of I/O and — especially with programs that use the Internet — network transactions. It's therefore valuable to know how to design network protocols for good performance.

The most important issue is avoiding protocol round trips as much as possible. Every protocol transaction that requires a handshake turns any latency in the connection into a potentially serious slowdown. Avoiding such handshakes is not specifically a Unix-tradition practice, but it's one that needs mention here because so many protocol designs lose huge amounts of performance to them.

> I cannot say enough about latency. X11 went well beyond X10 in avoiding round trip requests: the Render extension goes even further. X (and these days, HTTP/1.1) is a streaming protocol. For example, on my laptop, I can execute over 4 *million* 1×1 rectangle requests (8 million no-op requests) per second. But round trip requests are hundreds or thousands of times more expensive. Anytime you can get a client to do something without having to contact the server, you have a tremendous win.
>
> —
>
> <author>JimGettys</author>

In fact, a good rule of thumb is to design for the lowest possible latency and ignore bandwidth costs until your profiling tells you otherwise. Bandwidth problems can be solved later in development by tricks like compressing a protocol stream on the fly; but getting rid of high latency baked into an existing design is much, much harder (often, effectively impossible).

While this effect shows up most clearly in network protocol design, throughput vs. latency tradeoffs are a much more general phenomenon. In writing applications, you will sometimes face a choice between doing an expensive computation once in anticipation that it will be used several times, or computing only when actually needed (even if that means you will often recompute results). In most cases where you face a tradeoff like this, the right thing to do is bias toward low latency. That

is, don't try to precompute expensive operations unless you have a throughput requirement and know by actual measurement that the throughput you are getting is too low. Precomputation may seem efficient because it minimizes total use of processor cycles, but processor cycles are cheap. Unless you are doing one of a handful of monstrously compute-intensive applications like data mining, animation rendering, or the aforementioned bomb simulations, it is usually better to opt for short startup times and quick response.

In Unix's early days this advice might have been considered heretical. Processors were much slower and cost ratios were very different then; also, the pattern of Unix use was tilted rather more strongly toward server operations. The point about the value of low latency needs to be made partly because even newer Unix developers sometimes inherit an old-time cultural prejudice toward optimizing for throughput. But times have changed.

Three general strategies for reducing latency are (a) batching transactions that can share startup costs, (b) allowing transactions to overlap, and (c) caching.

Batching Operations

Graphics APIs are frequently written on the assumption that the fixed setup cost for a physical screen update is large. Consequently, the write operations actually modify an internal buffer. It is up to the programmer to decide when enough of these updates have been batched and to issue the call that turns them into a physical screen update. Picking the right spacing of physical updates can make a great deal of difference to the feel of the graphics client. Both the X server and the curses(3) library used by roguelike programs are organized in this way.

Persistent service daemons are a more Unix-specific example of batching. There are two reasons, one obvious and one subtle, to write persistent daemons (as opposed to CLI servers that are started up fresh for each session). The obvious reason is to manage updates to a shared resource. The less obvious reason, which obtains even for daemons that don't handle updates, is to amortize the cost of reading in the daemon's database across multiple requests. A perfect example of this is the DNS service daemon named(8), which must sometimes handle thousands of requests per second, each one of which may actually be blocking a user's Web page load. One of the tactics that makes named(8) fast is that it replaces parses of expensive on-disk text files describing DNS zones with accesses to a cache held in memory.

Overlapping Operations

In Chapter 5 we compared the POP3 and IMAP protocols for querying remote-mail servers. We noted that IMAP requests (unlike POP3 requests) are tagged with a request identifier generated by the client; the server, when it ships back a response, includes the tag of the request it pertains to.

POP3 requests have to be processed in lockstep by both client and server; the client sends a request, waits for the response to that request, and only then can prepare and ship the next one. IMAP requests, on the other hand, are are tagged so they can be overlapped. If an IMAP client knows that it wants to fetch multiple messages, it can stream several fetch requests (each with a different tag) to the IMAP server, without waiting for responses between them. Responses, each tagged, will come back when the server is ready; responses to early requests may come in while the client is still shipping later ones.

This strategy is general to more areas than network protocols. If you want to cut latency, blocking or waiting on intermediate results is deadly.

Caching Operation Results

Sometimes you can get the best of both worlds (low latency and good throughput) by computing expensive results as needed and caching them for later use. Earlier we mentioned that *named* reduces latency by batching; it also reduces latency by caching the results of previous network transactions with other DNS servers.

Caching has its own problems and tradeoffs, which are well illustrated by one application: the use of binary caches to eliminate parsing overhead associated with text database files. Some variants of Unix have used this technique to speed up access to their password information (the usual motivation was to cut latency on logins at very large sites).

To make this work, all code that looks at the binary cache has to know that it should check the timestamps on both files and regenerate the cache if the text master is newer. Alternatively, all changes to the textual master must be made through a wrapper that will update the binary format.

While this approach can be made to work, it has all the disadvantages that the SPOT rule would lead us to expect. The duplication of data means that it doesn't yield any economy of storage — it's purely a speed optimization. But the real problem with it is that the code to ensure coherency between cache and master is notoriously leaky and bug-prone. Very frequently updated cache files can lead to subtle race conditions simply because of the 1-second resolution of timestamps.

Coherency can be guaranteed in simple cases. One such is the Python interpreter, which compiles and deposits on disk a p-code file with extension `.pyc` when a Python library file is first imported. On subsequent runs the cached copy of the p-code is loaded unless the source has since changed (this avoids reparsing the library source code on every run). Emacs Lisp uses a similar technique with `.el` and `.elc` files. This technique works because both read and write accesses to the cache go through a single program.

When the update pattern of the master is more complex, however, the synchronization code tends to spring leaks. The Unix variants that used this technique to speed up access to critical system databases were infamous for spawning system-administrator horror stories that reflected this.

In general, binary cache files are a brittle technique and probably best avoided. The work that went into implementing a special-purpose hack to reduce latency in this one case would have been better spent improving the application design so it doesn't have a bottleneck there — or even on tuning to improve the speed of the file system or the virtual-memory implementation.

When you think you are in a situation that demands caching, it is wise to look one level deeper and ask why the caching is necessary. It may well be no more difficult to solve that problem than it would be to get all the edge cases in the caching software right.

Chapter 13. Complexity

As Simple As Possible, but No Simpler

Everything should be made as simple as possible, but no simpler.
--
<author>AlbertEinstein</author>

At the end of Chapter 1, we summarized the Unix philosophy as “Keep It Simple, Stupid!” Throughout the Design section, one of the continuing themes has been the importance of keeping designs and implementations as simple as possible. But what *is* “as simple as possible”? How do you tell?

We’ve held off on addressing this question until now because understanding simplicity is complicated. It needs some of the ideas we developed earlier in the Design section, especially in Chapter 4 and Chapter 11, as background.

The large questions in this chapter are central preoccupations of the Unix tradition, some of them motivating holy wars that have simmered for decades. This chapter starts from established Unix practice and vocabulary, then goes a bit further beyond it than we do in the rest of the book. We don’t try to develop simple answers to these questions, because there aren’t any — but we can hope that you will walk away with better conceptual tools for developing your own answers.

Speaking of Complexity

As with previous issues about modularity and interface design, Unix programmers react to a set of distinctions they have often learned from experience without knowing how to articulate. Therefore we’ll need to start by developing some terminology.

We will start by defining what software complexity is. We will make some horizontal distinctions between different flavors of complexity, which sometimes have to be traded off against each other. We will finish by making some even more important vertical distinctions, between the kinds of complexity we must live with and the kinds we have the option to eliminate.

The Three Sources of Complexity

Questions about simplicity, complexity, and the right size of software arouse a lot of passion in the Unix world. Unix programmers have learned a view of the world in which simplicity is beauty is elegance is good, and in which complexity is ugliness is grotesquery is evil.

Underlying the Unix programmer's passion for simplicity is a pragmatic fact: complexity costs. Complex software is harder to think about, harder to test, harder to debug, and harder to maintain — and above all, harder to learn and use. The costs of complexity, rough as they are during development, bite hardest after deployment. Complexity creates places for bugs to nest, from which they will emerge to trouble the world through the entire lifetime of their software.

All kinds of pressures tend to drag programmers into a swamp of complexity nevertheless. We've examined a rogue's gallery of these in earlier chapters; feature creep and premature optimization are the two most notorious. Traditionally, Unix programmers push back against these tendencies by proclaiming with religious fervor a rhetoric that condemns all complexity as bad.

So what exactly do we mean by 'complexity'? This point is worth pinning down, because it varies by observer.

Unix programmers (like other programmers) tend to focus on *implementation complexity*—basically, the degree of difficulty a programmer will experience in attempting to understand a program so he or she can mentally model or debug it.

Customers and users, on the other hand, tend to see complexity in terms of the program's *interface complexity*. In Chapter 11 we discussed the quality of ease and its inverse, mnemonic load. To a user, complexity correlates closely with mnemonic load. Poor expressiveness and concision can matter too, if a weak interface forces the user to perform lots of error-prone or merely tedious low-level operations rather than a few high-level ones.

Driven by both of these is a third measure that is much simpler: the total number of lines of code in the system, its *codebase size*. In terms of life-cycle costs, this is usually the most important measure. The reasons go back to perhaps the most important empirical result in software engineering, one we've cited before: the defect density of code, bugs per hundred lines, tends to be a constant independent of implementation language. More lines of code means more bugs, and debugging is the most expensive and time-consuming part of development.

Codebase size, interface complexity and implementation complexity may all rise together. That is the usual result of feature creep, and why programmers especially dread it. Premature optimization doesn't tend to raise interface complexity, but it has bad effects (often severely bad)

on implementation complexity and codebase size. But those sorts of arguments against complexity are relatively easy to win; the difficult ones begin when these three measures have to be traded off against each other.

We've already mentioned one situation in which two measures vary in opposite directions: a user interface that has been designed primarily to preserve implementation simplicity, or keep codebase size down, may simply dump low-level tasks on the user. (A crude example of this, barely imaginable to a Unix programmer but all too common elsewhere, might be an editor that lacked a global-replace feature.) Though this sort of design failure is all too common, it does not traditionally have a name. We'll call it a *manularity trap*.

Pressure to keep the codebase size down by using extremely dense and complicated implementation techniques can cause a cascade of implementation complexity in the system, leading to an undebuggable mess. This used to happen frequently when fitting programs onto very small systems demanded assembler programming or tricks like self-modifying code; nowadays it is uncommon except in embedded systems, and rapidly becoming rare even there. This kind of design failure doesn't have a traditional name, but one might call it a *blivet trap*, after an old Army term for the results of attempting to stuff ten pounds of horse manure into a five-pound bag.

The blivet trap won't appear in our case studies, but we've defined it for contrast with its opposite. It can happen that the designers of a project are so wary of implementation complexity that they reject a complex but unified way to solve a whole class of problems in favor of lots of duplicative, ad-hoc code that solves each individual one in turn. The result is bloat in the size of the codebase, and maintainability problems more severe than if the unified method had been accepted. For example, a Web project that really needs a centralized relational database behind its pages might instead spawn several different keyed data files containing information that has to be reintegrated at page generation time. This sort of failure is all too common. It doesn't have a traditional name; we'll call it an *adhocity trap*.

These are the three faces of complexity, and some of the traps designers fall into in attempts to avoid them.[114] We'll see more examples when we get to the case studies later in the chapter.

Tradeoffs between Interface and Implementation Complexity

One of the most perceptive observations ever made about the Unix tradition by someone standing outside it was contained in Richard Gabriel's paper called *Lisp: Good News, Bad News, and How to*

[114]The terms we have invented for these design traps, unlikely as they may sound, come from established hacker jargon described in [Raymond96].

Win Big [Gabriel]. Gabriel is a long-time leader of the Lisp community, and the paper was primarily an argument for a particular style of Lisp design, but the author himself acknowledges that it is now remembered primarily for the section called ‘The Rise of *Worse Is Better*’.

The paper argued that Unix and C have the characteristics of viruses, and that in the evolutionary struggle among software designs traits like implementation simplicity and portability which lead to rapid propagation (infectiousness) are more effective than correctness and completeness of the design. Gabriel came so close to anticipating the ‘many-eyeballs’ effect on open-source software that the open-source community retrospectively adopted him as one of its theorists after 1997.

Less remembered is that the Gabriel’s central argument was about a very specific tradeoff between implementation and interface complexity, one which rather exactly fits the categories we have examined in this chapter. Gabriel contrasts an ‘MIT’ philosophy most valuing interface simplicity with a ‘New Jersey’ philosophy most valuing implementation simplicity. He then proposes that although the MIT philosophy leads to software that is better in the abstract, the (worse) New Jersey model has better propagation characteristics. Over time, people pay more attention to software written in the New Jersey style, so it improves faster. Worse becomes better.

In fact, the MIT and New Jersey philosophies have analogs as conflicting tendencies within the Unix design tradition itself. One strain of Unix thinking emphasizes small sharp tools, starting designs from zero, and interfaces that are simple and consistent. This point of view has been most famously championed by Doug McIlroy. Another strain emphasizes doing simple implementations that work, and that ship quickly, even if the methods are brute-force and some edge cases have to be punted. Ken Thompson’s code and his maxims about programming have often seemed to lean in this direction.

The tension between these approaches arises precisely because one can sometimes get a simpler interface if one is willing to pay implementation complexity for it, or vice versa. Gabriel’s original example, about how system calls that do long operations handle interrupts they cannot hold or mask, is still one of the best. Under the MIT philosophy, the right thing to do would be to back out of the system call and automatically resume it once the interrupt has been handled; this is harder to implement but leads to a simpler interface. Under the New Jersey philosophy, the system call would return an error indicating that it has been interrupted and the user must re-execute; this can be implemented far more simply, but leads to a programming interface that is more difficult to use.

Both approaches have been tried. Old Unix hands will instantly think of System-V-style vs. BSD-style handling of software signals; the latter follows the MIT philosophy, while the former hails from New Jersey. Underlying the choice between them is a pressing question that has nothing directly to

do with the software's infectiousness: if your goal is to hold down total global complexity, where are you most willing to pay to do that? Where *should* you be most willing to pay?

One epochal example not mentioned in Gabriel's paper is from distributed hypertext systems. Early distributed-hypertext projects such as NLS and Xanadu were severely constrained by the MIT-philosophy assumption that dangling links were an unacceptable breakdown in the user interface; this constrained the systems to either browsing only a controlled, closed set of documents (such as on a single CD-ROM) or implementing various increasingly elaborate replication, caching, and indexing methods in an attempt to prevent documents from randomly disappearing. Tim Berners-Lee cut through this Gordian knot by punting the problem in classic New Jersey style. The simplicity of implementation he bought by allowing "404: Not Found" as a response was what made the World Wide Web lightweight enough to propagate and succeed.

Gabriel himself, while sticking with the observation that 'worse' is more infectious and tends to win in the end, has publicly changed his mind several times about the underlying complexity-related question of whether or not this is actually a good thing. His uncertainty mirrors a lot of ongoing design debates within the Unix community.

We cannot offer a one-size-fits-all answer. As with most of the large questions in this chapter, good taste and engineering judgement will demand different answers in different situations. The important thing is to develop the habit of thinking carefully about this issue on each and every one of your designs. As we have observed before in discussing software modularity, complexity is a cost you must budget very carefully.

Essential, Optional, and Accidental Complexity

In an ideal world, Unix programmers would craft only small, perfect gems of software, each minimal, each elegant, each perfect. But one of the unfortunate things about reality is that it often poses complex problems that demand complex solutions. You can't control a jetliner with an elegant ten-line procedure. There are too many pieces of equipment, too many channels and interfaces, too many different processors — too many different subsystems defined by independently operating human beings who often don't agree even on fundamental conventions. Even if you are successful at making all the individual software parts of an avionics system elegant, integration is likely to produce a large, complex, and grubby body of code with (one hopes) the single virtue that it will actually *work*.

Jetliners have *essential complexity*. There is a rather sharp point past which it's not possible to trade away features for simplicity, because the plane has to stay in the air. Because of that very

fact, avionics control systems do not tend to spawn religious wars about complexity — and Unix programmers tend to stay away from them.

Jetliners are certainly not immune from system failures due to overcomplexity. But the design issues are easier to discern and think about in software for which the requirements are more flexible, in which it is easy to trade off between anticipated features and complexity. (Here, and in the rest of this chapter, we will use ‘feature’ in a very general sense that includes things like performance gains or overall degree of interface polish.)

To sharpen our vision, we need to begin by noticing a difference between *accidental complexity* and *optional complexity*.[115] Accidental complexity happens because someone didn’t find the simplest way to implement a specified set of features. Accidental complexity can be eliminated by good design, or good redesign. Optional complexity, on the other hand, is tied to some desirable feature. Optional complexity can be eliminated only by changing the project’s objectives.

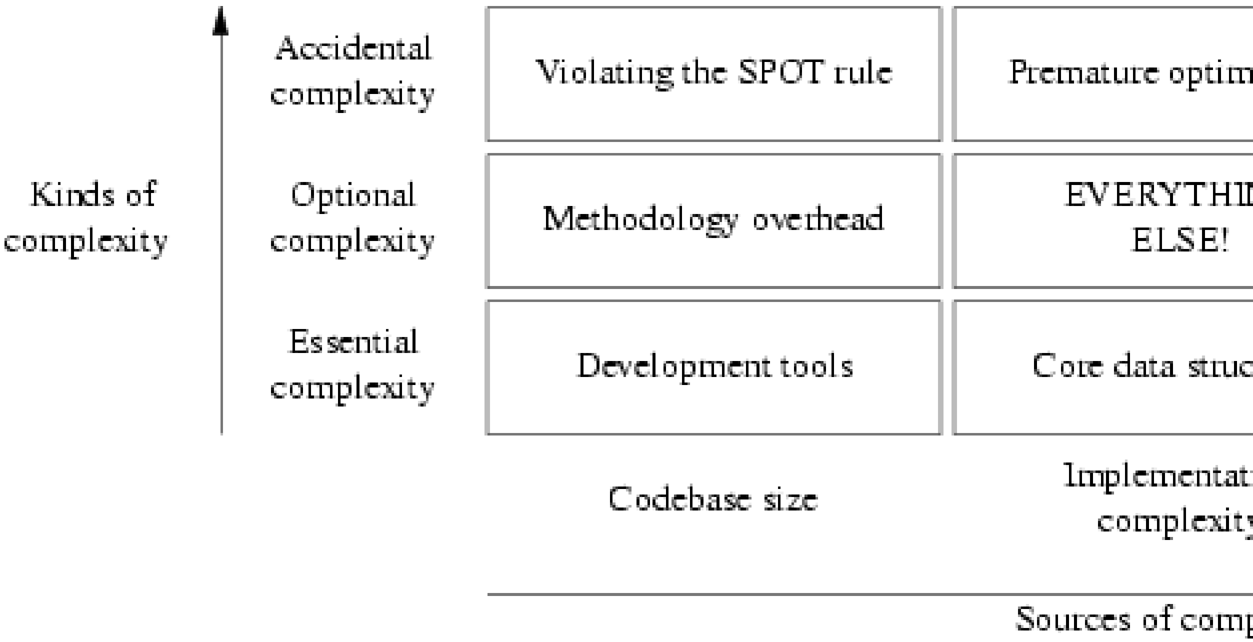
When we fail to distinguish between optional and accidental complexity, design debates become seriously confused. Questions about what a project’s objectives are get confused with questions about the aesthetics of simplicity, and whether people have been sufficiently clever.

Mapping Complexity

So far, we’ve developed two different scales for thinking about complexity. These scales are actually orthogonal to each other. Figure 13.1 may help clarify the relationships. Each of the nine boxes of the figure lists a common source of a particular kind of complexity.

[115] The distinction between accidental and optional complexity means that the categories we’re discussing here are *not* the same as essence and accident in Fred Brooks’s essay *No Silver Bullet* [Brooks], but they have common ancestry in philosophy.

Figure 13.1. Sources and kinds of complexity.



We've touched on some of these varieties of complexity earlier in this book, especially the accidental ones. In Chapter 4 we saw that accidental interface complexity often comes from non-orthogonality in the interface design — that is, failing to carefully factor the interface operations so that each does exactly one thing. Accidental code complexity (making code more complicated than it needs to be to get the job done) often results from premature optimization. Accidental codebase bloat often results from violating the SPOT rule, duplicating code or organizing it poorly so that opportunities for reuse aren't recognized.

Essential interface complexity usually can't be cut without trimming the basic functional requirements for the software (a theme we'll develop further in this chapter's case studies). Essential codebase size is related to choice of development tools because, if the feature list is held constant, the most important factor in codebase size is probably the choice of implementation language (as we implied in Chapter 8).

Sources of optional complexity are the most difficult to make useful generalizations about, because they so often depend on delicate judgments about which features it is worth paying the complexity cost for. Optional interface complexity often comes from adding convenience features that make life easier for users but aren't essential to the function of the program. Optional increases in codebase size (supposing the user-visible features and the algorithms used are held constant) can often come from various sorts of practices intended to make it more maintainable — adding mode comments, using long variable names, and so forth. Optional implementation complexity tends to be driven by *everything* that touches a project.

The sources of complexity have to be grappled with in different ways. Codebase size can be attacked with better tools. Implementation complexity can be addressed with better choice of algorithms. Interface complexity has to be addressed with better interaction design, a skill involving considerations of ergonomics and user psychology. This skill is less common (and possibly more difficult) than writing code.

Attacking the kinds of complexity, on the other hand, has to be done more with insight than with methods. You cut accidental complexity by noticing that there is a simpler way to do things. You cut optional complexity by making context-dependent judgments about what features are worthwhile. You can only cut essential complexity by having an epiphany, fundamentally redefining the problem you are addressing.

When Simplicity Is Not Enough

The failure mode that goes with the Unix tradition's insistence on simplicity is that Unix programmers often talk (and sometimes even behave) as though all optional complexity is accidental. More than this, there is a strong bias in the Unix tradition toward removing features rather than accepting optional complexity.

The case for this attitude is easy to make (indeed, we spend much of this book making it). Clean minimalism makes us feel virtuous on many levels, and designing for it is a valuable counter to the natural tendency of software systems to develop ever-more-elaborate encrustations of ill-considered features. But computing resources and human thinking time, like wealth, find their justification not in being hoarded but in being spent. As with other forms of asceticism, one has to ask when design minimalism stops being a valuable form of self-discipline and starts being a mere hair shirt — a way to indulge those feelings of virtue at the expense of actually *using* that wealth to get work done.

This is a perilous question, all too easily turned into an argument for abandoning good design discipline altogether. Unix old hands often shy away from it, fearing that failing to hold the

hardest possible line against complexity and bloat will lead us inexorably to damnation. But it's also a *necessary* question. We'll tackle it directly when analyzing this chapter's case studies.

A Tale of Five Editors

Now we're going to use five different Unix editors as case studies. It will be helpful to bear in mind a set of benchmark tasks as we examine these designs:

- *Plain-text editing.* Manipulating plain ASCII (or, in this internationalized age, perhaps Unicode) files with no structure known to the editor above byte level, or perhaps line level.
- *Rich-text editing.* Editing of text with attributes; these might include font changes, color, or other sorts of properties of text spans (such as being a hyperlink). Editors that can do this have to be able to translate between some presentation of the attributes in the user interface and some on-disk representation of the data (such as HTML, XML, or other rich-text formats.)
- *Syntax awareness.* An editor that is syntax-aware knows that input events have a grammar, and does things like automatically changing the indent level when it recognizes the beginning or end of a block scope in a programming language. Editors that are syntax-aware also commonly highlight syntax with colors or distinguished fonts.
- *Output parsing* of batch command output. The commonest case of this in the Unix world is running a C compilation from inside the editor, trapping the error messages, and then being able to step through the error locations without leaving the editor.
- *Interaction* with helper subprocesses that persist and maintain state between editor commands. This capability, when present, has powerful consequences:
    - It's possible to drive a version-control system from inside the editor, performing file checkins and checkouts without dropping out to a shell window or separate utility.
    - It's possible to front-end a symbolic debugger inside the editor, such that (for example) when the run stops on a breakpoint the appropriate file and line is automatically visited.

- It's possible to edit remote files within the editor, by having it recognize when a filename refers to another host (recognizing some syntax like `/user@host:/path/to-file`). Provided you have the right access, such an editor can automatically run a utility like scp(1) or ftp(1) to fetch a local copy, then automatically copy the edited version back to the remote location at file-save time.

All our case studies can edit plain text. (The reader should not take this capability for granted — there are many things called editors, such as 'word processors' that are too specialized to do this!) We begin seeing variable degrees of optional complexity in how they handle the more complex tasks.

ed

ed(1) is the truly Unix-minimalist way of plain-text editing. It dates from the days of teletypes.[116] It has a simple, austere CLI, and there is no screen display. In the following listing, computer output is *emphasized*.

```
ed sample.txt
sample.txt: No such file or directory
# This is a comment line, not a command.
# The message above warns that the sample.txt file is newly created.
a
the quick brown fox
jumped over the lazy dog
.
# That was an append command, which added text to the file.
# The dot on a line by itself terminated the append.
1s/f[a-z]x/dragon/
# On line 1, replace the first substring matching an f followed by a
# lowercase alphabetic followed by x with 'dragon'.  The
# substitute command accepts basic regular expressions.
1,$p
the quick brown dragon
jumped over the lazy dog
# Print all lines from 1 to the last.
w
51
# That wrote the file to disk. The 'q' command ends the
```

[116] Younger readers may not be aware that terminals used to print. On paper. Very slowly.

```
# editing session.
q
```

Unbelievable as it may seem to a modern reader, most of Unix's original code was written with this editor. The reader with DOS experience may recognize here the original on which *EDLIN* was (crudely) modeled.

If one defines the job of an editor simply as enabling the user to create and modify plain text files, ed(1) is entirely sufficient for the job. Importantly to the Unix view of design correctness, it does nothing else. Many old-school Unix programmers half-seriously maintain that all editors with more features than ed has are simply bloated — and a few still who seriously believe this.

Appropriately, *ed* was Ken Thompson's deliberate simplification of the earlier *qed*[RitchieQED] editor — which was very similar (and the first editor to use regular expressions in the characteristic Unix way) but had multiple-buffer capability that Ken deliberately discarded. He judged it not worth the additional complexity.

A notable characteristic of ed(1) and all its descendants is the object-operation format of its commands (the session example shows an explicit range on the 'p' command). There is a relatively powerful syntax for specifying line ranges, either numerically, or by regular-expression pattern match, or by special shorthands for the current and last line. Most editor operations can be applied to any range. This is a good example of orthogonality.

Nowadays, ed(1) is primarily used as a program-driven editing tool in scripts — a role to which editors with more elaborate modes of interactivity are unsuited. There is a close variant called ex(1) which adds a few useful interactivity features such as command prompts; it is occasionally useful in rare cases when editing must be done over a slow serial line, or in certain unusual crash-recovery situations where the library support needed to run other editors is not accessible. For these reasons, every Unix includes an *ed* implementation and most include *ex* as well.

The sed(1) stream editor mentioned in Chapter 9 is also closely related to ed; many of the basic commands are the same, though designed to be invoked through command-line switches rather than from standard input.

Almost all Unix programmers have strayed from the path of austerity and minimalist virtue enough to normally use editors that at least present a roguelike, screen-oriented interface. However, the fact that the religion of ed persists[117] says a great deal that is worth noting about the Unix mindset.

vi

The original vi(1) editor was the first attempt to bolt a visual, roguelike interface onto the command set of ed(1). Like ed, its commands are generally single keystrokes, and it is particularly well suited to use by touch-typists.

The original vi didn't have mouse support, editing menus, macros, assignable key bindings, or any form of user customization. In line with the religion of ed, vi's partisans considered the lack of these features a virtue. On this view, one of vi's most important virtues is that you can start editing immediately on a new Unix system without having to carry along your customizations or worrying that the default command bindings will be dangerously different from what you're used to.

One characteristic of vi that beginners tend to find frustrating is a result of its terse single-keystroke commands. It has a *moded* interface — you are either in command mode or in text-insertion mode. In text-insertion mode, the only commands that work are the ESC key for mode exit and (on newer versions) the cursor-movement keys. In command mode, typing text will be interpreted as commands and do odd (and probably destructive) things to your content.

On the other hand, one property of the command set that vi fans particularly tout is the object-operation format it inherited from ed. Most of the extended commands also operate in a natural way on any line range.

Over the years, vi has bulked up considerably. Modern versions add mouse support, editing menus, unlimited undo (the original vi could only undo the last command), multiple files in separate buffers, and customization with a run-control file. However, the use of run-control files is still unusual, and in contrast to Emacs, the use of embedded general-purpose scripting has never caught on. Instead, vi implementations have grown individual capabilities to do things, like syntax awareness of C code and output parsing of C compiler error messages, by adding C code to vi itself. Subprocess interaction is not supported.

[117] The religion of ed is exemplified by a famous Usenet posting which the reader may be able to find with a Web search for "Ed is the standard editor". While it is clearly intended as parody, it is by no means clear that the author was entirely joking. Most Unix hackers would read it as an example of "Ha ha, only serious".

Sam

The *Sam* editor[118] was written by Rob Pike at Bell Labs in the mid-1980s. Sam was designed for the Plan 9 operating system, which we'll survey in Chapter 20. While the Sam editor is not widely known outside the Labs, it's favored by many of the original Unix developers who went on to work on Plan 9, including Ken Thompson himself.

Sam is a fairly straightforward descendant of ed, remaining much closer to its parent than vi. Sam incorporates only two new concepts: a curses-style text display and text selection with the mouse.

Each Sam session has exactly one command window, and one or more text windows. Text windows edit text, and command windows accept ed-style editing commands. The mouse is used to move between windows, and to select text regions within text windows. This is a clean, orthogonal, modeless design that discards most of the interface complexity of vi.

Most commands operate by default on a select region that can be painted with a mouse drag operation. The select region for a command can also be set by specifying a line range in the fashion of ed, but Sam gains considerable power from the fact that the user can select at finer granularity than a line range. Because the mouse is available to do selections and rapidly change focus between buffers (including the command buffer), Sam needs no equivalent of the default (command) mode of vi. The hundreds of extended vi commands are unnecessary and, therefore, omitted. Overall, Sam adds only about a dozen commands to the seventeen or so in the ed set, for a total of about thirty.

Four of the new commands in Sam join two inherited from ed(1) and vi(1), as ways to apply regular expressions to the task of selecting files and file regions to operate on. These provide limited but effective loop and conditional facilities to the command language. There is, however, no way to name or parameterize command-language procedures. Nor can the language do interactive control of a subprocess.

An interesting feature of Sam is that it's split into two parts. separating a back end that manipulates files and does searches from a front end that handles the screen interface. This instance of the "separated engine and interface" chapter has the immediate practical benefit that, though the program has a GUI, it can run easily over a low-bandwidth connection to edit files on a remote server. Also, the front and back ends can be retargeted relatively easily.

Sam, like recent versions of vi, has infinite undo. By design, it supports neither rich-text editing, nor output parsing, nor subprocess interaction.

[118] http://plan9.bell-labs.com/sys/doc/sam/sam.html

Emacs

Emacs is undoubtedly the most powerful programmer's editor in existence. It's a big, feature-laden program with a great deal of flexibility and customizability. As we observed in the Chapter 14 section on Emacs Lisp, Emacs has an entire programming language inside it that can be used to write arbitrarily powerful editor functions.

Unlike vi, Emacs doesn't have interface modes; instead, commands are normally control characters or prefixed with an ESC. However, in Emacs it is possible to bind just about any key sequence to any command, and commands can be stock or customized Lisp programs.

Emacs can edit multiple files, each in a separate buffer, and supports moving text among the buffers. Versions running under X have native mouse support.

The Lisp programs bound to Emacs keystrokes can perform arbitrary text transformations on a buffer. This capability is heavily used, among other things to define syntax-aware and rich-text editing modes for dozens of different languages and markup formats (beginning with support and color highlighting of C code as in vi, but going way beyond that). Each mode is simply a library file of Lisp code that is loaded on demand.

Emacs Lisp programs can also interactively control arbitrary subprocesses. Some notable consequences of this capability were listed earlier, including the ability to serve as a front end for version-control systems, debuggers, and the like.

The designers of Emacs[119] built a programmable editor that could have task-related intelligence customized into it for hundreds of different specialized editing jobs. They then gave it the ability to drive other tools. As a result, Emacs supports dealing with all things textual in one shared context — files, mail, news, debugger symbols. It can serve as a customizable front end to any command with an interactive textual interface.

It is a common joke, both among fans and detractors of Emacs, to describe it as an operating system masquerading as an editor. That overstates the case, but Emacs certainly does fulfill the role occupied by integrated development environments (IDEs) under non-Unix operating systems (a theme to which we shall return in Chapter 15).

[119]The designers of Emacs were Richard M. Stallman, Bernie Greenberg, and Richard M. Stallman. The original Emacs was Stallman's invention, the first version with an embedded Lisp was Greenberg's, and the now-definitive version is Stallman's derived from Greenberg's. No complete account of the design history has been written in 2003, but Greenberg's *Multics Emacs: The History, Design, and Implementation* is illuminating and readily discoverable via keyword search on the Web.

This power comes at a price in complexity. To use a customized Emacs you have to carry around the Lisp files that define your personal Emacs preferences. Learning how to customize Emacs is an entire art in itself. Emacs is correspondingly harder to learn than vi.

Wily

The *wily* editor[120] is a clone of the Plan 9 editor *acme*.[121] It shares some facilities with Sam, but is intended to provide a fundamentally different user experience. Although Wily probably sees the least widespread use of any of these editors, it is interesting because it illustrates a different and arguably more Unixy way of implementing an Emacs-like programmable editor.

Wily could be described as a minimalist IDE, an implementation of Emacs-style extensibility without the decades of accompanying cruft. In Wily, even global search and replace, that *sine qua non* of Unix editors, is supplied by an external program. The built-in commands relate almost exclusively to windowing operations. Wily is designed from the ground up to use the mouse as much, and as well, as possible.

Wily attempts to replace not only conventional editors but conventional terminal windows such as xterm(1) as well. In Wily, any piece of text within the main window (which contains multiple non-overlapping Wily windows) can be an action or a search expression. The left mouse button is used to select text, the middle button to execute text as a command (either built-in or external), and the right button to search either Wily's buffers or the file system for text. No permanent or popup menus are required.

In Wily, the keyboard is used *only* to enter text. Shortcuts are achieved not by special use of the keyboard, but by holding down more than one mouse button at the same time. These shortcuts are always equivalent to using the middle button on some built-in command.

Wily can also be used as the front end for C, Python, or Perl programs, reporting to them whenever a window is changed or an execute or search command is performed with the mouse. These plugins function analogously to Emacs modes, but don't run in the same address space with Wily; instead, they communicate with it via a very simple set of remote procedure calls. Wily comes packaged with an *xterm* analog and a mail tool which uses it as the editing front end.

Because Wily depends on the mouse so heavily, it cannot be used on a character-cell-only console display; nor can it be used over a remote link without X forwarding. As an editor, Wily is

[120] http://www.cs.yorku.ca/~oz/wily

[121] http://plan9.bell-labs.com/sys/doc/acme/acme.html

designed for editing plain text; it has only two fonts (one proportional and one fixed-width) and has no mechanism that could support rich-text editing or syntax awareness.

The Right Size for an Editor

Now let us examine our case studies using the complexity categories we developed at the beginning of this chapter.

Identifying the Complexity Problems

Every text editor has a certain amount of essential complexity. At minimum, it has to maintain an internal buffer copy of the file or files the user is editing. Functions to import and export file data are a minimum requirement (usually from and to disk, though the stream editor sed(1) is an interesting exception). Some way to modify the buffer must be supported, though we cannot specify what way without describing specific features that are optional. Our four examples show widely varying levels of optional and accidental complexity beyond this.

Of all of these, ed(1) has the least complexity. Almost the only non-orthogonal feature in its command set is the fact that many of its commands can take a ‘p’ or ‘l’ suffix to print or list command results. Even after three decades of feature additions there are fewer than thirty editing commands, and the normal working set for most users will be less than a dozen. There is not much in the way of optional complexity that could be removed here, and it’s hard to identify any accidental complexity at all. The user interface of ed is strictly compact.

On the flip side, the ed interface is not really suitable for editing tasks even as basic as rapidly flipping through a text file. One has to limit one’s objectives pretty sharply for ed to become an acceptable solution for interactive editing.

Suppose, then, that we add “support visual browsing and editing of multiple files” as an objective? Then Sam seems not very far from being the minimal ed extension that could achieve this. The fact that the designers did not change the semantics of the inherited ed commands is notable; they kept an existing, orthogonal set and added a relatively small set of capabilities that are themselves orthogonal.

One large increase in optional (implementation) complexity is Sam’s infinite-undo capability. Another significant one is the new regular-expression-based loop and iteration facility in the command language. These, and the fact that the mouse can be used as a selection device, are about all that distinguish Sam from a hypothetical ed with a mouse-and-windows interface.

Without a thorough code audit it's difficult to be sure, but at the design level it's hard to identify any accidental complexity in Sam. The interface is at least semi-compact and arguably strictly compact. This editor lives up to the very highest standards of Unix design — unsurprisingly, given its provenance.

By contrast, vi looks rather bloated and flabby. There are hundreds of commands, many of them duplicative. These are at best optional complexity, and perhaps accidental. At a guess, most users don't know more than 5% of the command set. With the example of Sam before us, it's fair to wonder why the interface complexity of vi is so high.

In Chapter 11 we described the effect of the absence of standard arrow keys on early roguelike programs; vi was one of these. When vi was built, its author knew that many of his users would need to be able to use the cursor motion keys traditional on Unix glass teletypes. This made a modal interface inevitable. Once the hjkl keys had mode-dependent meanings in an edit buffer, it was all too easy to fall into the habit of adding new commands in an ad-hoc way.

Sam, designed as it is to depend on a bitmapped display with both arrow keys and a mouse, can be much cleaner. And it is.

But the clutter of vi commands is a relatively superficial problem. It's interface complexity, yes, but of a kind most users can and do ignore (the interface is semi-compact in the sense we developed in Chapter 4). The deeper problem is an adhocity trap. Over the years, vi has had progressively more and more special-purpose C code bolted onto it to perform tasks that Sam refuses to do and that Emacs would attack with Lisp code modules and subprocess control. The extensions are not, as in Emacs, libraries loaded as needed; users pay the overhead for the resulting code bloat all the time. As a result, the size difference between a modern vi and a modern Emacs is not nearly as great as one might expect; in mid-2003 on an Intel-architecture machine, it's 1500KB for GNU Emacs versus 900KB for vim. There is a whole lot of both optional and accidental complexity in that 900KB.

For vi partisans, not having an embedded scripting language — not being Emacs — has become an identity issue, a central part of the shared myth that vi is a lightweight editor. While vi fans like to talk about filtering buffers with external programs and scripts to do what Emacs's embedded scripting does, the reality is that vi's "!" command cannot filter regions of an edit buffer selected at finer granularity than a range of lines (Sam and Wily, though they have no more subprocess management than vi does, can at least filter arbitrary text ranges, not just line ranges). All knowledge of file formats and syntaxes that vary at a finer granularity (and most do) has to be built in to C code if vi is going to have it available at all. There is thus little prospect that the

codebase-size ratio between Emacs and vi will improve in favor of vi; indeed, it seems likely to get worse.

Emacs is sufficiently large, and has a sufficiently tangled history, to make separating its optional from its accidental complexity quite a challenge. We can at least begin by trying to separate the dispensable accidents of the Emacs design from its indispensable essentials.

Perhaps the most conspicuously dispensable part of the Emacs design is Emacs Lisp. It is essential to what Emacs does that it features what we nowadays call an embedded scripting language, but Emacs would be little different in capability if that language had been Python or Java or Perl. At the time Emacs was designed in the 1970s, however, Lisp was about the only language that had the characteristics (including unlimited-extent types and garbage collection) to fit it to the job.

Much in the particulars of the way *emacs* handles event processing and drives a bitmapped display (including the support for internationalization) is accidental as well. The one great schism in its history (the GNU Emacs/XEmacs fork) was over these issues, and demonstrates that nothing in the rest of the design prefers or requires any one event model.

On the other hand, the ability to bind arbitrary event sequences to arbitrary built-in or user-defined functions is indispensable. The scripting language could change and the event model could change, but without the anything-goes polymorphism in the way they are connected, the Emacs design would be both unrecognizable and crippled. Extension modes would have to fight each other for ownership of a limited event set, and activating multiple cooperating modes on the same buffer would be difficult or impossible.

The huge library of extension modes shipped with Emacs is accidental as well. The *ability* to construct such extensions may be essential, but the particular set we have is a product of history and chance. They could all be different or replaced; the result would still, recognizably, be Emacs.

But subprocess interaction is indispensable. Without it, Emacs modes could not perform the expected IDE-like integration and front-ending of many different tools.

Experience with small editors that clone the default keybindings and appearance of Emacs without emulating its extensibility is instructive. There have been several such clones, of which the best known are probably *MicroEmacs* and *pico*, but none have ever acquired significant mindshare.

Having identified accident and essence in the Emacs design helps us get a handle on which of its complexity is optional and which accidental. But, more importantly, they help us see past the superficial differences between Emacs and the previous three editors we have considered, to the

really critical difference: the fact that the objectives of the Emacs design are far more broad. Emacs wants to be a unified interface to all tools that operate on text.

Wily makes an interesting contrast with Emacs. As with Sam, the amount of optional complexity is low; the Wily user interface can be succinctly but effectively described in a single page.

But this elegance comes with a price; it is not possible to bind functions to any keystrokes or input gestures other than a restricted set of mouse chords. Instead, every editor function other than very basic text insertion and deletion has to be implemented with a program outboard of the editor, either a standalone script or a specialized symbiont process listening to Wily input events. (The former technique relies on outboard program startups being fast enough not to produce noticeable interface lag, something which was emphatically not the case in either Emacs's natal environment or under the Unixes it was first ported to.)

Optional complexity which Emacs would implement in Lisp extension modes is instead distributed through specialized symbionts; each has to know the special Wily messaging interface. An advantage of this approach is that such symbionts can be written in any language the user chooses. In addition, the symbionts (because they run outboard) cannot adversely affect each other or the Wily core (which is not true of Emacs modes). A disadvantage is that Wily itself cannot directly do subprocess interaction with ordinary Unix tools at all.

In this and other ways, *wily's* distributed scripting is not as powerful as the embedded scripting of Emacs. The scope of Wily's objectives is correspondingly narrower; the authors disclaim any interest in syntax-aware editing, or rich text, for example, and neither Wily nor its Plan 9 ancestor *acme* can do these things.

This brings us to another, and sharper way of posing the central question of this chapter: When do large objectives justify a large program?

Compromise Doesn't Work

The comparison between Sam and vi suggests strongly that, at least where editors are concerned, attempts to compromise between the minimalism of ed and the all-singing-all-dancing comprehensiveness of Emacs don't work very well; vi attempts this, and ends up with neither virtue. Instead, it falls into an adhocity trap. Wily avoids the adhocity trap, but cannot match the power of Emacs and must demand a custom process interface from each of its interactive symbionts in order to come anywhere close.

Evidently something about editors tends to push them in the direction of increasing complexity. In the case of vi, that something is not hard to identify; it's the desire for convenience. While ed may be theoretically adequate, very few people (other than perhaps Ken Thompson himself) would forgo screen-oriented editing to make a statement about software bloat.

More generally, programs that mediate between the user and the rest of the universe notoriously attract features. This includes not just editors but Web browsers, mail and newsgroup readers, and other communications programs. All tend to evolve in accordance with the Law of Software Envelopment, aka Zawinski's Law: "Every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can".

Jamie Zawinski, inventor of the Law (and one of the principal authors of the Netscape and Mozilla Web browsers), maintains more generally that all really useful programs tend to turn into Swiss Army knives. The commercial success of large, integrated application suites outside the Unix world tends to confirm this, and directly challenges the Unix philosophy of minimalism.

To the extent Zawinski's Law is correct, it suggests that some things want to be small and some want to be large, but the middle ground is unstable. The superficial problems with vi can be put down to history, but the deeper ones trace back to the combination of steady pressure to add features with refusal to embed the scripting and subprocess-control features that vi partisans associate with excessive size. On a different level, accepting that there would be two modes in the interface (insertion versus character-motion) opened a can of worms — it became far too easy to add new commands without thinking about their complexity impact on the overall design.

The examples of Emacs and Wily further suggest *why* some things want to be large: so that several related tasks can share context. Editing and version control (or editing and mail, editing and symbolic debugging, etc.) are separate tasks from the point of view of the implementers — but users would often prefer to have one big environment that lets them point at pieces of text, rather than spend time and attention ping-ponging between several programs that each have to have the same filename or the contents of some cut buffer handed to them.

More generally, let's suppose we view the entire Unix environment as a single work of design by community. Then the religion of "small, sharp tools", the pressure to keep interface complexity and codebase size down, may lead right to a manularity trap — the user has to maintain all the shared context himself, because the tools won't do it for him.

Returning to the specific context of editors, Sam shows us that vi is the wrong thing. Wily is a valiant effort to avoid the vastness of Emacs that falls short because it can't be syntax-aware. But

Wily, or some realization of the Emacs design ideas cleaned up and stripped of historical baggage, might be the right thing. The value of optional complexity depends on the objectives you choose, and the ability to share context among all the text-oriented tools related to a task is valuable.

Is Emacs an Argument against the Unix Tradition?

The traditional Unix view of the world, however, is so attached to minimalism that it isn't very good at distinguishing between the adhocity-trap problems of vi and the optional complexity of Emacs.

> The reason that vi and emacs never caught on among old-school Unix programmers is that they are *ugly*. This complaint may be "old Unix" speaking, but had it not been for the singular taste of old Unix, "new Unix" would not exist.
>
> —
>
> <author>DougMcIlroy</author>

Attacks on Emacs by vi users — along with attacks on vi by the hard-core old-school types still attached to ed — are episodes in a larger argument, a contest between the exuberance of wealth and the virtues of austerity. This argument correlates with the tension between the old-school and new-school styles of Unix.

The "singular taste of old Unix" was partly a consequence of poverty in exactly the same way that Japanese minimalism was — one learns to do more with less most effectively when having more is not an option. But Emacs (and new-school Unix, reinvented on powerful PCs and fast networks) is a child of wealth.

> As, in a different way, was old-school Unix. Bell Labs had enough resources so that Ken was not confined by demands to have a product yesterday. Recall Pascal's apology for writing a long letter because he didn't have enough time to write a short one.
>
> —
>
> <author>DougMcIlroy</author>

Ever since, Unix programmers have maintained a tradition that exalts the elegant over the excessive.

The vastness of Emacs, on the other hand, did not originate under Unix, but was invented by Richard M. Stallman within a very different culture that flourished at the MIT Artificial Intelligence Lab in the 1970s. The MIT AI lab was one of the wealthiest corners of computer-science academia; people learned to treat computing resources as cheap, anticipating an attitude that would not be

viable elsewhere until fifteen years later. Stallman was unconcerned with minimalism; he sought the maximum power and scope for his code.

The central tension in the Unix tradition has always been between doing more with less and doing more with more. It recurs in a lot of different contexts, often as a struggle between designs that have the quality of clean minimalism and others that choose expressive range and power even at the cost of high complexity. For both sides, the arguments for or against Emacs have exemplified this tension since it was first ported to Unix in the early 1980s.

Programs that are both as useful and as large as Emacs make Unix programmers uncomfortable precisely because they force us to face the tension. They suggest that old-school Unix minimalism is valuable as a discipline, but that we may have fallen into the error of dogmatism.

There are two ways Unix programmers can address this problem. One is to deny that large is actually large. The other is to develop a way of thinking about complexity that is not a dogma.

Our thought experiment with replacing Lisp and the extension libraries gives us a new perspective on the oft-heard charge that Emacs is bloated because its extension library is so large. Perhaps this is as unfair as charging that */bin/sh* is bloated because the collection of all shellscripts on a system is large. Emacs could be considered a virtual machine or framework around a collection of small, sharp tools (the modes) that happen to be written in Lisp.

On this view, the main difference between the shell and Emacs is that Unix distributors don’t ship all the world’s shellscripts along with the shell. Objecting to Emacs because having a general-purpose language in it feels like bloat is approximately as silly as refusing to use shellscripts because shell has conditionals and for loops. Just as one doesn’t have to learn shell to use shellscripts, one doesn’t have to learn Lisp to use Emacs. If Emacs has a design problem, it’s not so much the Lisp interpreter (the framework part) as the fact that the mode library is an untidy heap of historical accretions — but that’s a source of complexity users can ignore, because they won’t be affected by what they don’t use.

This mode of argument is very comforting. It can be applied to other tool-integration frameworks, such as the (uncomfortably large) GNOME and KDE desktop projects. There is some force to it. And yet, we should be suspicious of any ‘perspective’ that offers to resolve all our doubts so neatly; it might be a rationalization, not a rationale.

Therefore, let’s avoid the possibility of falling into denial and accept that Emacs is both useful and large — that it *is* an argument against Unix minimalism. What does our analysis of the kinds of

complexity in it, and the motives for it, suggest beyond that? And is there reason to believe that those lessons generalize?

The Right Size of Software

There is a hidden dual of the Unix gospel of small, sharp tools; a background so implicit that many Unix practitioners do not notice it, any more than fish notice the water they swim in. It is the presence of frameworks.

Small, sharp tools in the Unix style have trouble sharing data, unless they live inside a framework that makes communication among them easy. Emacs is such a framework, and *unified management of shared context* is what the optional complexity of Emacs is buying. The practical impact of unified management of shared context is that the user is not burdened with low-level naming and resource-management issues.

In old-school Unix, the only framework was pipelines, redirection, and the shell; the integration was done with scripts, and the shared context was (essentially) the file system itself. But that was not the end of evolution.

Emacs unifies the file system with a world of text buffers and helper processes, largely leaving the shell framework behind. Wily is also about buffers and helpers, but incorporates the shell framework into itself. Modern desktop environments provide a communication framework for GUIs, also leaving the shell framework behind. Each framework has strengths and weaknesses of its own. Frameworks become homes to ecologies of tools — the shell to shellscripts, Emacs to Lisp modes, and desktop environments to flocks of GUIs communicating both via drag and drop and by more esoteric means such as object brokers.

This suggests a Rule of Minimality: *Choose the shared context you want to manage, and build your programs as small as those boundaries will allow.* This is “as simple as possible, but no simpler”, but it focuses attention on the choice of shared context. It applies not just to frameworks, but to applications and program systems.

It is, however, all too easy to get sloppy about how large your shared context needs to be. The pressure behind Zawinski’s Law is the tendency of applications to want to share context for convenience. It’s easy to end up carrying around too much weight, too many assumptions, and to write programs that are over-complex, bloated, and huge. The paradigmatic example in the 1990s was the way that the mailto: URL induced the growth of huge mail clients embedded in Web browsers.

The corrective to this tendency comes straight from the old-school Unix hymnbook. It is the Rule of Parsimony: *Write a big program only when it is clear by demonstration that nothing else will do*—that is, when attempts to partition the problem have been made and failed. This maxim implies an astringent skepticism about large programs, and a strategy for avoiding them: look for the small-program solution first. If a single small program won't do the job, try building a toolkit of cooperating small programs within an existing framework to attack it. Only if both approaches fail are you free (in the Unix tradition) to build a large program (or a new framework) without feeling you have failed the design challenge.

When you do write a framework, remember the Rule of Separation. Frameworks should be mechanism, and have as little policy as possible. In most cases, that is no policy at all. Factor as much behavior as possible into modules that use the framework. One of the benefits of writing or reusing a framework is that it can help you separate what would otherwise be big lumps of policy into separate modules, modes, or tools — pieces that can be usefully recombined with others.

These rules are valuable heuristics, but the tension at the heart of the Unix tradition does not resolve neatly into a set of *a-priori* prescriptions for optimal size of any given project. Circumstances alter cases, and exercising good judgment and good taste is what software designers are for. As in Soto Zen, the journey *is* the destination; enlightenment has to be rediscovered in every day of practice.

Part III. Implementation

Chapter 14. Languages

To C or Not To C?

The limits of my language are the limits of my world.
--
<author>LudwigWittgenstein</author>
Tractatus Logico-Philosophicus 5.6, 1918

Unix's Cornucopia of Languages

Unix supports a wider variety of application languages than does any other single operating system; indeed, it may well have hosted more different languages than every other operating system in the history of computing combined.[122]

There are at least two excellent reasons for this huge diversity. One is the wide use of Unix as a research and teaching platform. The other (far more relevant for working programmers) is the fact that matching your application design with the proper implementation language(s) can make an immense difference in your productivity. Therefore the Unix tradition encourages the design of domain-specific languages (as we mentioned in Chapter 7 and Chapter 9) and what are now generally called *scripting languages*—those designed specifically to glue together other applications and tools.

> The term "scripting language" probably derives from the term "script" that was applied to a potted input for a normally interactive program, in particular sh or ed — a much more felicitous term than the "runcom" we inherited from Unix's ancestor CTSS. "Script" appears in the V7 manual (1979). I don't recall who coined the name.
>
> —
> <author>DougMcIlroy</author>

In truth, the term 'scripting language' is a somewhat awkward one. Many of the the major languages usually so described (Perl, Tcl, Python, etc.) have outgrown the group's scripting origins and are now standalone general-purpose programming languages of considerable power. The term tends to obscure strong similarities in style with other languages that are not usually lumped in with this group, notably Lisp and Java. The only argument for continuing to use it is that nobody has yet invented a better term.

[122] See the Free Compiler and Interpreter List [ftp://ftp.idiom.com/pub/compilers-list/free-compilers] for details.

Part of the reason all these can be lumped together under the rubric of ‘scripting language’ is that they all have pretty much the same ontogeny. Having a runtime to do interpretation also makes it relatively easy to automate dynamic storage management. Automating dynamic storage management almost requires using references (opaque memory addresses that you can’t do arithmetic on) rather than passing value copies or explicit pointers around. Using references makes runtime polymorphism and OO an easy next step. *Voila:* the modern scripting language!

To apply the Unix philosophy effectively, you’ll need to have more than just C in your toolkit. You’ll need to learn how to use some of Unix’s other languages (especially the scripting languages), and how to be comfortable mixing multiple languages in specialist roles within large program systems.

In this chapter we’ll survey C and its most important alternatives, discussing their strengths and weaknesses and the sorts of tasks to which they are best matched. The languages covered will be C, C++, shell, Perl, Tcl, Python, Java, and Emacs Lisp. Each survey section will include case studies on applications written using these languages, and references to other examples and tutorial material. High-quality implementations of all these languages are available in open source on the Internet.

Warning: Choice of application language is one of the archetypal religious issues in the Internet/Unix world. People get very attached to these tools and will sometimes defend them past all reason. If this chapter achieves its aim, zealots of all stripes may be offended by this chapter, but everyone else will learn from it.

Why Not C?

C is the native language of Unix. Since the early 1980s it has come to dominate systems programming almost everywhere in the computer industry. Outside of Fortran’s dwindling niche in scientific and engineering computing, and excluding the vast invisible dark mass of COBOL financial applications at banks and insurance companies, C and its offspring C++ have now (in 2003) dominated applications programming almost completely for more than a decade.

It may therefore seem perverse to assert that C and C++ are nowadays almost always the wrong vehicle for beginning new applications development. But it’s true; C and C++ optimize for machine efficiency at the expense of increased implementation and (especially) debugging time. While it still makes sense to write system programs and time-critical kernels of applications in C or C++, the world has changed a great deal since these languages came to prominence in the 1980s. In 2003,

processors are a thousand times faster, memories are a thousand times larger, and disks are a factor of *ten* thousand larger, for roughly constant dollars.[123]

These plunging costs change the economics of programming in a fundamental way. Under most circumstances it no longer makes sense to try to be as sparing of machine resources as C permits. Instead, the economically optimal choice is to minimize debugging time and maximize the long-term maintainability of the code by human beings. Most sorts of implementation (including application prototyping) are therefore better served by the newer generation of interpreted and scripting languages. This transition exactly parallels the conditions that, last time around the wheel, led to the rise of C/C++ and the eclipse of assembler programming.

The central problem of C and C++ is that they require programmers to do their own memory management — to declare variables, to explicitly manage pointer-chained lists, to dimension buffers, to detect or prevent buffer overruns, and to allocate and deallocate dynamic storage. Some of this task can be automated away by unnatural acts like retrofitting C with a garbage collector such as the Boehm-Weiser implementation, but the design of C is such that this cannot be a complete solution.

C memory management is an enormous source of complication and error. One study (cited in [Boehm]) estimates that 30% or 40% of development time is devoted to storage management for programs that manipulate complex data structures. This did not even include the impact on debugging cost. While hard figures are lacking, many experienced programmers believe that memory-management bugs are the single largest source of persistent errors in real-world code.[124] Buffer overruns are a common cause of crashes and security holes. Dynamic-memory management is particularly notorious for spawning insidious and hard-to-track bugs, such as memory leaks and stale-pointer problems.

Not so long ago, manual memory management made sense anyway. But there are no ‘small systems’ any more, not in mainstream applications programming. Under today’s conditions, an implementation language that automates away memory management (and buys an order of magnitude decrease in bugs at the expense of using a bit more cycles and core) makes a lot more sense.

[123]Outside the Unix world, this three-orders-of-magnitude improvement in hardware performance has been masked to a significant extent by a corresponding drop in software performance.

[124]The severity of this problem is attested to by the rich slang Unix programmers have developed for describing different varieties: ‘aliasing bug’, ‘arena corruption’, ‘memory leak’, ‘buffer overflow’, ‘stack smash’, ‘fandango on core’, ‘stale pointer’, ‘heap trashing’, and the rightly dreaded ‘secondary damage’. See the Jargon File [http://www.catb.org/~esr/jargon] for elucidation.

A recent paper [Prechelt] musters an impressive array of statistical evidence for a claim that programmers with experience in both worlds will find very plausible: programmers are just about twice as productive in scripting languages as they are in C or C++. This accords well with the 30%–40% penalty estimate noted earlier, plus debugging overhead. The performance penalty of using a scripting language is very often insignificant for real-world programs, because real-world programs tend to be limited by waits for I/O events, network latency, and cache-line fills rather than by the efficiency with which they use the CPU itself.

The Unix world has been slowly coming around to this point of view in practice, especially since 1990 or so, as is shown by the increasing popularity of Perl and other scripting languages. But the evolution of practice has not yet (as of mid-2003) led to a wholesale change in conscious attitudes; many Unix programmers are still absorbing the lesson Perl and Python have been teaching.

We can see the same trend happening, albeit more slowly, outside the Unix world — for example, in the continuing shift from C++ to Visual Basic evident in applications development under Microsoft Windows and NT, and the move toward Java in the mainframe world.

The arguments against C and C++ apply with equal force to other conventional compiled languages such as Pascal, Algol, PL/I, FORTRAN, and compiled Basic dialects. Despite occasional heroic efforts such as Ada, the differences between conventional languages remain superficial when set against their basic design decision to leave memory management to the programmer. Though high-quality open-source implementations of most languages ever written are available under Unix, no other conventional languages remain in widespread use in the Unix or Windows worlds; they have been abandoned in favor of C and C++. Accordingly we will not survey them here.

Interpreted Languages and Mixed Strategies

Languages that avoid manual memory management do it by having a memory manager built into their runtime executable somewhere. Typically, runtime environments in these languages are split into a program part (the running script itself) and the interpreter part, with the interpreter managing dynamic storage. On Unixes (and other modern operating systems) the interpreter core can be shared by multiple program parts, reducing the effective overhead for each one.

Scripting is nowhere near a new idea in the Unix world. As far back as the mid-1970s, in an era of far smaller machines, the Unix shell (the interpreter for commands typed to a Unix console) was designed as a full interpreted programming language. It was common even then to write programs entirely in shell, or to use the shell to write glue logic that knit together canned utilities and custom programs in C into wholes greater than the sum of their parts. Classical introductions to the Unix

environment (such as *The Unix Programming Environment* [Kernighan-Pike84]) have dwelt heavily on this tactic, and with good reason: it was one of Unix's most important innovations.

Advanced shell programming mixes languages freely, employing both binaries and interpreted elements from half a dozen or more other languages for subtasks. Each language does what it does best, each component is a module with narrow interfaces to the others, and the global complexity of the whole is much lower than it would be had it been coded as a single monster monolith in a general-purpose language.

Language Evaluations

Mixing languages is a knowledge-intensive (rather than coding-intensive) style of programming. To make it work, you have to have both working knowledge of a suitable variety of languages and expertise about what they're best at and how to fit them together. In this section, we will try to point you at references to help you with the first and an overview to convey the second. For each language surveyed we will include case studies of successful programs that exemplify its strengths.

C

Despite the memory-management problem, there are some application niches for which C is still king. Programs that require maximum speed, have real-time requirements, or are tightly coupled to the OS kernel are good candidates for C.

Programs that must be portable across multiple operating systems may also be good candidates for C. Some of the alternatives to C that we shall discuss below are, however, increasingly penetrating major non-Unix operating systems; in the near future, portability may be less a distinguishing advantage of C.

Sometimes the leverage to be gained from existing programs like parser generators or GUI builders that generate C code is so great that it justifies C coding of the rest of a small application.

And, of course, C proved indispensable to the developers of all its alternatives. Dig down through enough implementation layers under any of the other languages surveyed here and you will find a core implemented in pure, portable C. These languages inherit many of the advantages of C.

Under modern conditions, it's perhaps best to think of C as a high-level assembler for the Unix virtual machine (recall the discussion of the success of C as a case study in Chapter 4). C standards have exported many of the facilities of this virtual machine, such as the standard I/O library, to other

operating systems. C is where you go when you want to get as close as possible to the bare metal but stay portable.

One good reason to learn C, even if your programming needs are satisfied by a higher-level language, is that it can help you learn to think at hardware-architecture level. The best reference and tutorial on C for people who are already programmers is still *The C Programming Language* [Kernighan-Ritchie].

Porting C code between Unix variants is almost always possible and usually easy, but specific areas of variation (like signals and process control) can be tricky to get right. We highlight some of these issues in Chapter 17. Differing C bindings on other operating systems can of course cause C portability problems, although Windows NT at least theoretically supports an ANSI/POSIX-compliant C API.

High-quality C compilers are available as open-source software over the Internet; the best-known and most widely used is the Free Software Foundation's GNU C compiler (part of GCC, the GNU Compiler Collection), which has become the native C of all open-source Unix systems and many even in the closed-source world. GCC ports are even available for Microsoft's family of operating systems. GCC sources are available at the FSF's FTP site [ftp://ftp.gnu.org/pub/gnu].

Summing up: C's best side is resource efficiency and closeness to the machine. Its worst side is that programming in it is a resource-management hell.

C Case Study: *fetchmail*

The best case study for C is the Unix kernel itself, for which a language that naturally supports hardware-level operations is actually a strong advantage. But *fetchmail* is a good example of the kind of user-land utility that is still best coded in C.

fetchmail does only the simplest kind of dynamic-memory management; its only complex data structure is a singly-linked list of per-mailserver control blocks built just once, at startup time, and changed only in fairly trivial ways afterwards. This substantially erodes the case against using C by sidestepping C's greatest weakness.

On the other hand, these control blocks are fairly complex (including all of string, flag, and numeric data) and would be difficult to handle as coherent fast-access objects in an implementation language without an equivalent of the C struct feature. Most of the alternatives to C are weaker than C in this respect (Python and Java being notable exceptions).

Finally, *fetchmail* requires the ability to parse a fairly complex specification syntax for per-mail-server control information. In the Unix world this sort of thing is classically handled by using C code generators that grind out source code for a tokenizer and grammar parser from declarative specifications. The existence of *yacc* and *lex* was a point in favor of C.

fetchmail might reasonably have been coded in Python, albeit with possibly significant loss of performance. Its size and data-structure complexity would have excluded shell and Tcl right off and strongly counterindicated Perl, and the application domain is outside the natural scope of Emacs Lisp. A Java implementation wouldn't have been an unreasonable path, but Java's object-oriented style and garbage collection would have offered little purchase on *fetchmail*'s specific problems over what C already yields. Nor could C++ have done much to simplify the relatively simple internal logic of *fetchmail*.

However, the real reason *fetchmail* is a C program is that it evolved by gradual mutation from an ancestor already written in C. The existing implementation has been extensively tested on many different platforms and against many odd and quirky servers. Carrying all that implicit knowledge through to a re-implementation in a different language would be messy and difficult. Furthermore, *fetchmail* depends on imported code for functions (like NTLM authentication) that don't seem to be available above C level.

fetchmail's interactive configurator, which did not have a C legacy problem, is written in Python; we'll discuss that case along with that language.

C++

When C++ was first released to the world in the mid-1980s object-oriented (OO) languages were being widely touted as the silver bullet for the software-complexity problem. C++'s OO features appeared to be an overwhelming advantage over the ancestral C, and partisans expected that it would rapidly make the older language obsolete.

This has not happened. Part of the fault can be laid to problems in C++ itself; the requirement that it be backward-compatible with C forced a great many compromises on the design. Among other things, that requirement prevented C++ from going to fully automatic dynamic-memory management and addressing C's most serious problem. Later, feature arms races between different compiler implementers, unconstrained by a weak and premature standardization effort, pushed C++ to become rather baroque and excessively complicated.

Another part of the fault must be laid to the failure of OO itself to live up to expectations. We examined this problem in Chapter 4, observing the tendency of OO methods to lead to thick glue layers and maintenance problems. Today (2003), inspection of open-source archives (in which choice of language reflects developers' judgments rather than corporate mandates) reveals that C++ usage is still heavily concentrated in GUIs, multimedia toolkits and games (the major success areas for OO design), and little used elsewhere.

It may be that C++'s realization of OO is particularly problem-prone. There is some evidence that C++ programs have higher life-cycle costs than equivalents in C, FORTRAN, or Ada. Whether this is a problem with OO or specifically with C++ or both remains unclear, though there is reason to suspect both are implicated [Hatton98].

In recent years, C++ has incorporated some important non-OO ideas. It has exceptions similar to those in Lisp; that is, it is possible to throw an object or value up the call stack until it is caught by a handler. STL (Standard Template Library) provides generic programming; that is, it is possible to code algorithms that are independent of the type signature of their data and have them compiled to do the right thing at runtime. (Only languages that do compile-time static type-checking need this; more dynamic languages simply pass around typeless references and support type identification at runtime.)

Efficient compiled language; upward-compatible with C; object-oriented platform; vehicle for cutting-edge techniques like STL and generics — C++ tries to be all things to all people, but the cost is more complexity than the mind of any individual programmer can handle. As we noted in Chapter 4, the language's principal designer has conceded that he doesn't expect any one programmer to grasp it all. Unix hackers do not react well to this; one anonymous but famous characterization is "C++: an octopus made by nailing extra legs onto a dog".

When all is said and done, however, C++'s most fundamental problem is that it is basically just another conventional language. It confines the memory-management problem better than it did before the invention of the Standard Template Library, and a lot better than C does, but the confinement is brittle; it breaks unless your code uses objects and only objects. For many types of application its OO features are not significant, and simply add complexity to C without yielding much advantage. Open-source C++ compilers are available; if C++ were unequivocally superior to C it would now dominate.

Summing up: C++'s best side is its combination of compiled efficiency with facilities for OO and generic programming. Its worst side is that it is baroque and complex, and tends to encourage over-complex designs.

Consider using C++ if an existing C++ toolkit or service library offers powerful leverage for your application, or if you're in one of the application areas mentioned above for which an OO language is known to be a large win.

The classic C++ reference is Stroustrup's *The C++ Programming Language* [Stroustrup]. You will find an excellent beginner's tutorial on C++ and basic OO methods in *C++: A Dialog* [Heller]. *C++ Annotations* [Brokken] is a condensed introduction to C++ for expert C programmers.

The Gnu Compiler Collection includes a C++ compiler. The language is therefore universally available on Unix and on Microsoft operating systems; comments made under C above also apply here. Strong collections of open-source support libraries [http://www.boost.org/] are available. However, portability is compromised by the fact that (as of mid-2003) actual C++ implementations implement widely varying subsets of the draft ISO standard now in preparation.[125]

C++ Case Study: The Qt Toolkit

The Qt interface toolkit is one of the notable C++ success stories in today's open-source world. It provides a widget set and API for writing graphical user interfaces under X, one deliberately (and rather effectively) designed to emulate the visual look and feel of Motif, MacOS Platinum, or the Microsoft Windows interface. Qt actually provides more than just GUI services; it also provides a portable application layer, with classes for XML, file access, sockets, threads, timers, time/date handling, database access, various abstract data types, and Unicode.

The Qt toolkit is a critical and visible component of the KDE project, the senior of the open-source world's two efforts to produce a competitive GUI and integrated set of desktop productivity tools.

Qt's C++ implementation exhibits the strengths of an OO language for encapsulating user-interface components. In a language supporting objects, a visual hierarchy of interface widgets can be cleanly expressed in the code by a hierarchy of class instances. While this sort of thing can be simulated in C with explicit indirection through hand-rolled method tables, such code is much cleaner in C++. Comparison with the notoriously baroque C API of Motif is instructive.

The Qt source code and reference documentation are available at the Trolltech site [http://www.trolltech.com/].

Shell

[125] The last C++ standard, dating from 1998, was widely implemented but weak, especially in the area of libraries.

The ‘Bourne shell’ (*sh*) of Version 7 Unix was Unix’s first (and for many years its only) portable interpreted language. Today the ancestral Bourne shell has largely been displaced by variants of the upward-compatible Korn Shell (*ksh*); the single most important of these is the Bourne Again Shell, bash.

A few other shells exist and are used interactively, but are not significant as programming languages; of these, the best known is probably the C shell *csh*, which is notoriously[126] unsuitable for writing scripts.

Simple shell programs are extremely easy and natural to write. The Unix tradition of rapid prototyping in interpretive languages began with shell.

> I wrote the very first version of netnews as a 150-line shellscript. It had multiple newsgroups and cross-posting; newsgroups were directories and cross-posting was implemented as multiple links to the article. It was far too slow to use for production, but the flexibility permitted endless experimentation with the protocol design.
>
> —
>
> <author>StevenM. Bellovin</author>

As program size gets larger, however, they tend to become rather ad-hoc. Some parts of shell syntax (notably its quoting and statement-syntax rules) can be very confusing. These drawbacks generally relate to compromises in the programming-language part of the shell’s design made to preserve its utility as an interactive command-line interpreter.

Programs are described as being ‘in shell’ even when they are not pure shell but include heavy use of C filters like sort(1) and of standard text-processing minilanguages like sed(1) or awk(1). This sort of programming has been in decline for some years, however; nowadays such elaborate glue logic is generally written in Perl or Python, with shell being reserved for the simplest kinds of wrappers (for which these languages would be overkill) and system boot-time initialization scripts (which cannot assume they are available).

Such basic shell programming should be adequately covered in any introductory Unix book. *The Unix Programming Environment* [Kernighan-Pike84] remains one of the best sources on intermediate and advanced shell programming. Korn shell implementations or clones are present on every Unix.

[126] See Tom Christiansen’s essay *Csh Programming Considered Harmful*, which should be readily findable via Web search.

Complex shellscripts often have portability problems, not so much because of the shell itself but because they make assumptions about what other programs are available as components. While Bourne and Korn-shell clones have been sporadically available on non-Unix operating systems, shell programs are not (practically speaking) at all portable off Unix.

Summing up: shell's best side is that it is very natural and quick for small scripts. Its worst side is that large shellscripts depend on lots of auxiliary commands that aren't necessarily identically behaved nor even present on all target machines. Nor is it easy to analyze the dependencies in a large shellscript.

It is almost never necessary to build or install a shell, since all Unix systems and Unix emulators come equipped with them. The standard shell on Linux and other leading-edge Unix variants is now *bash*.

Case Study: *xmlto*

xmlto is a driver script that calls all the commands needed to render an XML-DocBook document as HTML, PostScript, plain text, or in any one of several other formats (we'll take a closer look at DocBook in Chapter 18). It is written in bash.

xmlto handles the details of calling an XSLT engine with appropriate stylesheet, then handing off the result to a postprocessor. For HTML and XHTML the XSLT transformation does the entire job. For plain text, the XML is also processed into HTML, but then handed to a postprocessor — lynx(1) in its `-dump` mode, which renders HTML to flat text. For PostScript, the XML is transformed to XML FO (formatting objects) which a postprocessor then massages into *TeX* macros, to DVI format via tex(1), and then finally to PostScript via the well-known dvi2ps(1) tool.

xmlto consists of a single front-end shellscript. It calls any one of several script plugins, each named after the target format. Each plugin is a shellscript. Depending on how it's called, it either supplies a stylesheet for the front end to apply, or calls the appropriate postprocessor(s) with various canned arguments.

This architecture means that all the information about a given output format lives in one place (the corresponding script plugin), so adding new output types can be done without disturbing the front-end code at all.

xmlto is a good example of a medium-sized shell application. Neither C nor C++ would have made sense because they are awkward for scripting. Any of the other scripting languages in this chapter

could have been used for this job; but it's all simple command dispatching, with no internal data structures or complex logic, so shell is good enough. Shell has the significant advantage of being ubiquitous on the intended target systems.

In theory this script could run on any system supporting *bash*. The real constraint is the requirement for one of the XSLT engines and all the postprocessors needed to be present on the system. In practice, this script is not likely to run anywhere but under one of the modern open-source Unixes.

Case Study: Sorcery Linux

Sorcerer GNU/Linux is a Linux distribution that you install as a small, bootable foothold system just powerful enough to run bash(1) and a couple of download utilities. With this code in place, you can invoke Sorcery, the Sorcerer package system.

Sorcery handles installing, removing, and integrity-checking software packages. When you "cast spells", Sorcery downloads the source code, compiles it, installs it, and saves a list of files that were installed (along with a build log and checksums for all the files). Installed packages can be "dispelled" or removed. Package listing and integrity checks are also available. More details are available at the Sorcery project site [http://sorcerer.wox.org].

The Sorcery system is written entirely in shell. Program installation procedures tend to be small, simple programs for which shell is appropriate. In this particular application, the main drawback of shell is neutralized because Sorcery's authors can guarantee that the helper programs they need will be present in the foothold system.

Perl

Perl is shell on steroids. It was specifically designed to replace awk(1), and expanded to replace shell as the 'glue' for mixed-language script programming. It was first released in 1987.

Perl's strongest point is its extremely powerful built-in facilities for pattern-directed processing of textual, line-oriented data formats; it is unsurpassed at this. It also includes far stronger data structures than shell, including dynamic arrays of mixed element types and a 'hash' or 'dictionary' type that supports convenient and fast lookup of name-value pairs.

Additionally, Perl includes a rather complete and well-thought-out internal binding of virtually the entire Unix API, drastically reducing the need for C and making it suitable for jobs like simple

TCP/IP clients and even servers. Another strong advantage of Perl is that a large and vigorous open-source community has grown up around it. Its home on the net is the Comprehensive Perl Archive Network [http://www.cpan.org]. Dedicated Perl hackers have written hundreds of freely reusable Perl modules for many different programming tasks. These include everything from structure-walking of directory trees through X toolkits for GUI building, through excellent canned facilities for supporting HTTP robots and CGI programming.

Perl's main drawback is that parts of it are irredeemably ugly, complicated, and must be used with caution and in stereotyped ways lest they bite (its argument-passing conventions for functions are a good example of all three problems). It is harder to get started in Perl than it is in shell. Though small programs in Perl can be extremely powerful, careful discipline is required to maintain modularity and keep a design under control as program size increases. Because some limiting design decisions early in Perl's history could not be reversed, many of the more advanced features have a fragile, klugey feel about them.

The definitive reference on Perl is *Programming Perl* [Wall2000]. This book has nearly everything you will ever need to know in it, but is notoriously badly organized; you will have to dig to find what you want. A more introductory and narrative treatment is available in *Learning Perl* [Schwartz-Christiansen].

Perl is universal on Unix systems. Perl scripts at the same major release level tend to be readily portable between Unixes (provided they don't use extension modules). Perl implementations are available (and even well documented) for the Microsoft family of operating systems and on MacOS as well. Perl/Tk provides cross-platform GUI capability.

Summing up: Perl's best side is as a power tool for small glue scripts involving a lot of regular-expression grinding. Its worst side is that it is ugly, spiky, and nigh-unmaintainable in large volumes.

A Small Perl Case Study: *blq*

The *blq* script is a tool for querying block lists (lists of Internet sites that have been identified as habitual sources of unsolicited bulk email, aka spam). You can find current sources at the *blq* project page [http://www.unicom.com/sw/blq/].

blq is a good example of a small Perl script, illustrating both the strengths and weaknesses of the language. It makes intensive use of regular-expression matching. On the other hand, the Net::DNS

Perl extension module it uses has to be conditionally included, because it is not guaranteed to be present in any given Perl installation.

blq is exceptionally clean and disciplined as Perl code goes, and I recommend it as an example of good style (the other Perl tools referenced from the *blq* project page are good examples as well). But parts of the code are unreadable unless you are familiar with very specific Perl idioms — the very first line of code, **$0 =~ s!.*/!!;**, is an example. While all languages have some of this kind of opacity, Perl has it worse than most.

Tcl and Python are both good for small scripts of this type, but both lack the Perl convenience features for regular-expression matching that *blq* uses heavily; an implementation in either would have been reasonable, but probably less compact and expressive. An Emacs Lisp implementation would have been even faster to write and more compact than the Perl one, but probably painfully slow to use.

A Large Perl Case Study: *keeper*

keeper is the tool used to file incoming packages and maintain both FTP and WWW index files for the huge Linux free-software archives at ibiblio. You can find sources and documentation in the search tools subdirectory of the ibiblio archive [http://www.ibiblio.org].

keeper is a good example of a medium-to-large interactive Perl application. The command-line interface is line-oriented and patterned after a specialized shell or directory editor; note the embedded help facilities. The working parts make heavy use of file and directory handling, pattern matching, and pattern-directed editing. Note the ease with which *keeper* generates Web pages and electronic-mail notifications from programmatic templates. Note also the use of a canned Perl module to automate walking various functions over directory trees.

At about 3300 lines, this application is probably pushing the size and complexity limit of what one should attempt in a single Perl program. Nevertheless, most of it was written in a period of six days. In C, C++, or Java it would have taken a minimum of six weeks and been extremely difficult to debug or modify after the fact. It is way too large for pure Tcl. A Python version would probably be structurally cleaner, more readable, and more maintainable — but also more verbose (especially near the pattern-matching parts). An Emacs Lisp mode could readily do the job, but Emacs is not well suited for use over a telnet link that is often slowed to a crawl by server congestion.

Tcl

Tcl (Tool Command Language) is a small language interpreter designed to link with compiled C libraries, providing scripted control of C code (*extended scripts*). Its original application was to control libraries for electronic simulators (SPICE-like applications). Tcl is also suitable for *embedded scripts*—that is, scripts called from within C programs and returning values to those programs. Tcl had its first general public release in 1990.

Some facilities built on top of Tcl have achieved wide use outside the Tcl community itself. The two most important of these are:

- The Tk toolkit, a kinder and gentler X interface that makes it easy to rapidly build buttons, dialog boxes, menu trees, and scrolling text widgets and collect input from them.
- Expect, a language that makes it relatively easy to script fully interactive programs with widely variable responses.

The Tk toolkit is so important that the language is often referred to as Tcl/Tk. Tk is also frequently used with Perl and Python.

The main advantage of Tcl itself is that it is extremely flexible and radically simple. The syntax is very odd (based on a positional parser) but totally consistent. There are no reserved words, and there is no syntactic distinction between a function call and 'built in' syntax; thus the Tcl language interpreter itself can be effectively redefined from within Tcl (which is what makes projects like *Expect* reasonable).

The main drawback of Tcl is that the pure language has only weak facilities for namespace control and modularity, and two of them (**upvar** and **uplevel**) are rather dangerous if not used with great caution. Also, there are no data structures other than association lists. Tcl therefore scales up very poorly — it is difficult to organize and debug pure Tcl programs of even moderate size (more than a few hundred lines) without tripping over your own feet. In practice, almost all large Tcl programs use one of several OO extensions to the language.

The oddities of the syntax can at first be a problem as well; the distinction between string quotes and braces will probably give you headaches for a while, and the rules for when things need to be quoted or braced are a bit tricky.

Pure Tcl only provides access to a relatively small and commonly used part of the Unix API (essentially just file handling, process-spawning, and sockets). Indeed, Tcl has the flavor of an

experiment in seeing how small a scripting language can get and still be useful. Tcl extensions (similar to Perl modules) provide a richer set of capabilities, but are (like CPAN modules) not guaranteed to be installed everywhere.

The original Tcl reference is *Tcl and the Tk Toolkit* [Osterhout94], but that book has been largely superseded by *Practical Programming in Tcl and Tk* [Welch]. Brian Kernighan has written a description of a real-world Tcl project [Kernighan95] that summarizes Tcl's strengths and weaknesses as a rapid-prototyping and production tool; his contrast with Microsoft Visual Basic is particularly balanced and instructive.

The Tcl world doesn't have one central repository run by a core group analogous to Perl's or Python's, but several excellent websites both point to each other and cover most Tcl tool and extension development. Look at the Tcl Developer Xchange [http://www.tcltk.com] first; among other things, it offers Tcl sources of an interactive Tcl tutorial. There is also a Tcl foundry at SourceForge [http://sourceforge.net/foundry/tcl-foundry/].

Tcl scripts have portability problems similar to those of shell scripts; the language itself is highly portable, but the components it calls may not be. Tcl implementations exist for the Microsoft family of operating systems, MacOS, and many other platforms. Tcl/Tk scripts will run on any platform with GUI capabilities.

Summing up: Tcl's best side is its spare, compact design and the extensibility of the Tcl interpreter. Its worst side is the odd positional parser and the weakness of its data structures and namespace controls; the latter defect makes it scale poorly for large projects.

Case Study: *TkMan*

TkMan is a browser for Unix man pages and Texinfo documents. At roughly 1200 lines, it is quite large to be written in pure Tcl, but the code is unusually well-modularized and mature. It uses Tk to provide a GUI interface quite a bit nicer than either the stock man(1) or xman(1) utilities support.

TkMan makes a good case study because it exhibits almost the full gamut of Tcl techniques. Highlights include Tk integration, scripted control of other Unix applications (such as the Glimpse search engine), and the use of Tcl to parse Texinfo markup.

Any of the other languages would have made for a less direct interface to the Tk GUI that constitutes most of this code.

A Web search for “TkMan” should turn up sources and documentation.

Moodss: A Large Tcl Case Study

The Moodss system is a graphical monitoring application for system administrators. It can watch logs and gather statistics for MySQL, Linux, SNMP networks, and Apache, and presents a digested view of them through spreadsheet-like GUI panels called 'dashboards'. Monitoring modules can be written in Python or Perl as well as Tcl. The code is polished, mature, and considered an exemplar in the Tcl community. There is a project website [http://jfontain.free.fr/moodss/].

The Moodss core consists of about 18,000 lines of Tcl. It uses several Tcl extensions including a custom object system; the Moodss author admits that without these "writing such a big application would not have been possible".

Again, any of the other languages would have made for a less direct interface to the Tk GUI that constitutes most of this code.

Python

Python is a scripting language designed for close integration with C. It can both import data from and export data to dynamically loaded C libraries, and can be called as an embedded scripting language from C. Its syntax is rather like a cross between that of C and the Modula family, but has the unusual feature that block structure is actually controlled by indentation (there is no analog of explicit begin/end or C curly brackets). Python was first publicly released in 1991.

The Python language is a very clean, elegant design with excellent modularity features. It offers designers the option to write in an object-oriented style but does not force that choice (it can be coded in a more classically procedural C-like way). It has a type system comparable in expressive power to Perl's, including dynamic container objects and association lists, but less idiosyncratic (actually, it is a matter of record that Perl's object system was built in imitation of Python's). It even pleases Lisp hackers with anonymous lambdas (function-valued objects that can be passed around and used by iterators). Python ships with the Tk toolkit, which can be used to easily build GUI interfaces.

The standard Python distribution includes client classes for most of the important Internet protocols (SMTP, FTP, POP3, IMAP, HTTP) and generator classes for HTML. It is therefore very well suited to building protocol robots and network administrative plumbing. It is also excellent for Web CGI work, and competes successfully with Perl at the high-complexity end of that application area.

Of all the interpreted languages we describe, Python and Java are the two most clearly suited for scaling up to large complex projects with many cooperating developers. In many ways Python is simpler than Java, and its friendliness to rapid prototyping may give it an edge over Java for standalone use in applications that are neither hugely complex nor speed critical. An implementation of Python in Java, designed to facilitate mixed use of these two languages, is available and in production use; it is called Jython.

Python cannot compete with C or C++ on raw execution speed (though using a mixed-language strategy on today's fast processors probably makes that relatively unimportant). In fact it's generally thought to be the least efficient and slowest of the major scripting languages, a price it pays for runtime type polymorphism. Beware of rejecting Python on these grounds, however; most applications do not actually need better performance than Python offers, and even those that appear to are generally limited by external latencies such as network or disk waits that entirely swamp the effects of Python's interpretive overhead. Also, by way of compensation, Python is exceptionally easy to combine with C, so performance-critical Python modules can be readily translated into that language for substantial speed gains.

Python loses in expressiveness to Perl for small projects and glue scripts heavily dependent on regular-expression capability. It would be overkill for tiny projects, to which shell or Tcl might be better suited.

Like Perl, Python has a well-established development community with a central website [http://www.python.org] carrying a great many useful Python implementations, tools and extension modules.

The definitive Python reference is *Programming Python* [Lutz]. Extensive on-line documentation on Python extensions is also available at the Python website.

Python programs tend to be quite portable between Unixes and even across other operating systems; the standard library is powerful enough to significantly cut the use of nonportable helper programs. Python implementations are available for Microsoft operating systems and for MacOS. Cross-platform GUI development is possible with either Tk or two other toolkits. Python/C applications can be ‘frozen’, quasi-compiled into pure C sources that should be portable to systems with no Python installed.

Summing up: Python's best side is that it encourages clean, readable code and combines accessibility with scaling up well to large projects. Its worst side is inefficiency and slowness, not just relative to compiled languages but relative to other scripting languages as well.

A Small Python Case Study: *imgsizer*

Imgsizer is a utility that rewrites WWW pages so that image-inclusion tags get the right image size parameters automatically plugged in (this speeds up page loading on many browsers). You can find sources and documentation in the URL WWW tools subdirectory of the ibiblio archive [http://www.ibiblio.org].

imgsizer was originally written in Perl, and was a nearly ideal example of the sort of small, pattern-driven text-processing tool at which Perl excels. It was later translated to Python to take advantage of Python's library support for HTTP fetching; this eliminated a dependency on an external page-fetching utility. Observe the use of file(1) and ImageMagick identify(1) as specialist tools for extracting the pixel sizes of images.

The dynamic string handling and sophisticated regular-expression matching required would have made *imgsizer* quite painful to write in C or C++; that version would also have been much larger and harder to read. Java would have solved the implicit memory-management problem, but is hardly more expressive than C or C++ at text pattern matching.

A Medium-Sized Python Case Study: *fetchmailconf*

In Chapter 11 we examined the *fetchmail*/*fetchmailconf* pair as an example of one way to separate implementation from interface. Python's strengths are well illustrated by *fetchmailconf*.

fetchmailconf uses the Tk toolkit to implement a multi-panel GUI configuration editor (Python bindings also exist for GTK+ and other toolkits, but Tk bindings ship with every Python interpreter).

In expert mode, the GUI supports editing of about sixty attributes divided among three panel levels. Attribute widgets include a mix of checkboxes, radio buttons, text fields, and scrolling listboxes. Despite this complexity, the first fully-functional version of the configurator took me less than a week to design and code, counting the four days it took to learn Python and Tk.

Python excels at rapid prototyping of GUI interfaces, and (as *fetchmailconf* illustrates) such prototypes are often deliverable. Perl and Tcl have similar strengths in this area (including the Tk toolkit, which was written for Tcl) but are hard to control at the complexity level (approximately 1400 lines) of *fetchmailconf*. Emacs Lisp is not suited for GUI programming. Choosing Java would have increased the complexity overhead of this programming task without delivering significant benefits for this nonspeed-intensive application.

A Large Python Case Study: PIL

PIL, the Python Imaging Library, supports the manipulation of bitmap graphics. It supports many popular formats, including PNG, JPEG, BMP, TIFF, PPM, XBM, and GIF. Python programs can use it to convert and transform images; supported transformations include cropping, rotation, scaling, and shearing. Pixel editing, image convolution, and color-space conversions are also supported. The PIL distribution includes Python programs that make these library facilities available from the command line. Thus PIL can be used either for batch-mode image transformation or as a strong toolkit over which to implement program-driven image processing of bitmaps.

The implementation of PIL illustrates the way Python can be readily augmented with loadable object-code extensions to the Python interpreter. The library core, implementing fundamental operations on bitmap objects, is written in C for speed. The upper levels and sequencing logic are in Python, slower but much easier to read and modify and extend.

The analogous toolkit would be difficult or impossible to write in Emacs Lisp or shell, which don't have or don't document a C extension interface at all. Tcl has a good C extension facility, but PIL would be an uncomfortably large project in Tcl. Perl has such facilities (Perl XS), but they are ad-hoc, poorly documented, complex, and unstable by comparison to Python's and use of them is rare. Java's Native Method Interface appears to provide a facility roughly comparable to Python's; PIL would probably have made a reasonable Java project.

The PIL code and documentation is available at the project website [http://www.pythonware.com/products/pil/].

Java

The Java programming language was designed to be "write once, run anywhere" and to support embedding interactive programs (or *applets*) in Web pages that would be runnable from any browser. Thanks to a series of technical and strategic blunders by its owner, Sun Microsystems, it has failed in both its original objectives. But it is still sufficiently strong at both systems and applications programming to be seriously challenging C and C++. Java was announced in 1995.

Java is cleverly designed to capture the huge benefit of automatic memory management and the lesser but not insignificant benefit of supporting OO design, while being far smaller and simpler than C++. It retains a broadly C-like syntax that most programmers will find comfortable. It includes support for callouts to dynamically-loaded C and calling Java as an embedded language from C. Nor is it trivial that Sun has done an excellent job of making good Java documentation available on the Web.

Against Java, we can say that (compared to, say, Python) some parts of it appear over-complex and others deficient. Java's class-visibility and implicit-scoping rules are baroque. The interface facility avoids complex problems with multiple inheritance at the cost of being only slightly less difficult to understand and use in itself. Features like inner and anonymous classes can lead to very confusing code. The absence of reliable destructor methods means that it is difficult to ensure proper management of resources other than memory, such as mutexes and file locks. Significant parts of the Unix operating-system facilities are not accessible from stock Java, including signals, poll, and select. While Java's I/O facilities are very powerful, simple reading of text files is not simple.

There is a particularly invidious problem, resembling Windows DLL hell, with libraries. Java has no method to manage different library versions. This can create huge problems in environments like application servers, where the server might come equipped with one version of (say) an XML library, but the application ships with a different (usually newer) version. The only handle on such problems is the `CLASSPATH` environment variable, a source of chronic deployment problems.

Furthermore, Sun's handling of the Java language has been both politically and technically obtuse. Java's first GUI toolkit, AWT, was a mess that had to be essentially replaced. Withdrawing the language from ECMA/ISO standardization further nettled many developers already upset by features of the Sun Community Source License (SCSL). Restrictions in the SCSL continue to hamper open-source implementations of Java 1.2 and their J2EE (Java 2 Enterprise Edition) specification. This compromises Java's original objective of universal portability.

Sadly, browser applets are dead. Microsoft's decision not to support Java 1.2 in Internet Explorer effectively killed them. However, Java seems to have found a secure niche in the computing ecology, for 'servlets' running within Web application servers. It has also become commonly used for a lot of in-house corporate programming not directly tied to databases or webservers. It has become major competition for both Microsoft's ASP/COM platform and Perl CGIs. Finally, it is in widespread and increasing use as a language for teaching introductory programming (a role for which it is extremely well suited).

Overall, we can fairly judge Java to be superior to C++ (which is both far more complex and does less to attack the memory-management problem) for all but systems programming and the most speed-critical applications. Experience seems to show that Java programmers are somewhat less likely to fall into the trap of excessive OO layering than are C++ programmers, though this remains a significant problem.

How Java will fare in equilibrium with the other languages we describe here is unclear as yet, and may depend largely on project scale. We may expect its proper niche to resemble Python's. Like

Python, it cannot compete with C or C++ on raw execution speed, nor against Perl on small projects that use pattern-driven editing heavily. It is (more definitely than Python) overkill for small projects. We may guess that Python will have an edge in smaller projects and Java in larger ones, but the verdict of experience is not yet in.

The best single reference on paper is probably *Java In A Nutshell* [FlanaganJava], but this is not the best tutorial introduction; that would probably be *Thinking in Java* [Eckel]. Trails to all the world's Java websites begin at Sun's Java site [http://java.sun.com], which also has complete HTML documentation available for download for free. The Open Directory Java Page [http://dmoz.org/Computers/Programming/Languages/Java/] also collects useful Java links.

Java implementations are available for all Unixes, for Microsoft operating systems, MacOS, and many other platforms.

Sources for Kaffe, an open-source Java implementation with class libraries conforming to most of JDK 1.1 and portions of JDK 1.2, are available at the Kaffe project site [http://www.kaffe.org/].

There is a Java front end for GCC. GCJ can compile Java code to either Java bytecode or native code, and can compile Java bytecode to native code as well. It comes packaged with open-source class libraries that implement most of JDK 1.2, and a Java bytecode interpreter called *gij*. Details are at the GCJ project page [http://gcc.gnu.org/java/].

There is a Java IDE for Emacs at the JDEE project site [http://jdee.sunsite.dk/].

Java portability is excellent at the language level. Incomplete library implementations (especially older JDK 1.1 versions that don't support the newer JDK 1.2) can be an issue.

Java's best side is that it comes close enough to achieving write-once-run-anywhere to be useful as an OS-independent environment of its own. Its worst side is that the Java 1/Java 2 split compromises that goal in deeply frustrating ways.

Case Study: FreeNet

Freenet is a peer-to-peer networking project that is intended to make censorship and content suppression impossible.[127] Freenet developers envision the following applications:

[127] There is a Freenet project website [http://freenetproject.org].

- Uncensorable dissemination of controversial information: Freenet protects freedom of speech by enabling anonymous and uncensorable publication of material ranging from grassroots alternative journalism to banned exposés.
- Efficient distribution of high-bandwidth content: Freenet's adaptive caching and mirroring is being used to distribute Debian Linux software updates.
- Universal personal publishing: Freenet enables anyone to have a website, without space restrictions or compulsory advertising, even if the would-be webmaster doesn't own a computer.

Freenet addresses these goals by providing a virtual space in which to publish documents that is not tied to any specific machine. Published information and Freenet's own internal data indexes are replicated and distributed across the network in such a way that even Freenet administrators don't know at any given time where all the physical copies are located. Privacy for people browsing or submitting to Freenet is protected by strong cryptography.

Java was a good choice for this project for at least two reasons. First: the goals of the project put a heavy premium on having compatible implementations on the widest possible variety of machines, so Java's high portability is a dominating advantage. Second: the nature of the project is such that the network API is important, and Java has a strong one built in.

C is traditional for infrastructure projects of this kind that have high performance demands, but the lack of a standardized network API would have made porting a significant difficulty. C++ would have had the same difficulty. Tcl, Perl, or Python might have reduced the porting burden, but at a greater cost in performance. Emacs Lisp would have been painfully slow and totally inappropriate.

Emacs Lisp

Emacs Lisp is a scripting language used to program the behavior of the Emacs text editor. Its first public release was in 1984.

Emacs Lisp is not a general-purpose language in quite the same way as the others surveyed in this chapter; while it is powerful enough to theoretically be used as such, it is traditionally employed only to write control programs for the Emacs editor itself and does not communicate as fluently with other software as would a modern scripting language.

Nevertheless, there is a significant range of applications in which Emacs Lisp is more effective than anything else. Many of these have to do with providing a front-end for development tools such as

the C compiler and linker, make(1), version-control systems, and symbolic debuggers; we'll discuss these in Chapter 15.

More generally, Emacs is to pattern- or syntax-directed *interactive* editing what Perl is to pattern-directed *batch* editing. Any application that involves interactively hacking a special file format or text database is an excellent candidate to be prototyped (and possibly delivered) as an Emacs mode (an Emacs Lisp program that specializes the editor's behavior).

Emacs Lisp is also valuable for building applications that have to be closely integrated with a text editor, or that function primarily as text browsers with some editing capability. User agents for email and Usenet news fall in this category. So do certain kinds of database front ends.

Emacs Lisp is a Lisp. It follows as the night the day that it manages memory automatically and is far more elegant and powerful than most conventional languages, or indeed most *unconventional* languages; it can compete with Java or Python on this level and laugh at C or C++, Perl, shell or Tcl. Lisp's perennial problem of lacking a standardized OS binding for portability is solved by the Emacs core, which in effect *is* its OS binding.

Lisp's other perennial problem — being a resource hog — is no longer a real issue on modern machines. Parody expansions like 'Emacs Makes A Computer Slow' and 'Eventually Munches All Computer Storage' used to be common (in fact the Emacs distribution itself includes a list of them). But many other commonly used categories of programs (such as Web browsers) have nowadays grown larger and more complex than Emacs, which has come to appear rather moderate by comparison.

The definitive Emacs Lisp reference is *The GNU Emacs Lisp Reference Manual*, which may be browseable through your Emacs's 'info' help system. If not, it can be downloaded from the FSF FTP site [ftp://ftp.gnu.org/pub/gnu]. If you find that impenetrable, *Writing GNU Emacs Extensions* [Glickstein] may help.

Portability of Emacs Lisp programs is excellent. Emacs implementations are available for all Unixes, the Microsoft operating systems, and Mac OS.

Summing up: Emacs Lisp's best point is that it combines an excellent base language, Lisp, with powerful domain primitives for text manipulation. Its worst point is poor performance and difficulties using it in communication with other programs.

For more information, see the discussion of *Emacs* under editors in the next chapter.

Trends for the Future

Table 14.1 gives a rough indication of today's distribution of language usage. We give figures from both SourceForge[128] and Freshmeat,[129] the two most important new-release sites, as of March 2003.

The SourceForge figures are soft in several ways: Notably, SourceForge's query interface doesn't permit filtering on OS and language simultaneously, so some of these numbers represent MacOS and Windows projects. The effect is probably to exaggerate C++ and Java's share considerably. However, Unix-based projects dominate sufficiently (by about a 3:1 ratio) so that the effect on the figures for languages other than these is probably not too distorting.

The Freshmeat sample is smaller, but the site hosts only Unix-based releases — and it counts actual releases, not the huge clutter of failed and inactive SourceForge projects. It is thus interesting that the population figures track SourceForge's by about a 1:2 ratio except in precisely the cases (C++ and Java) where we would expect them to be out of proportion because of the absence of Windows projects.

Table 14.1. Language choices.

Language	SourceForge	Freshmeat
C	10296	4845
C++	9880	2098
Shell	1058	487
Perl	4394	2508
Tcl	649	328
Python	2222	948
Java	8032	1900
Emacs Lisp	?	31

This chapter was first drafted in 1997; at time of writing it is mid-2003. That is a long enough time base that the relative positions of the languages we surveyed above have changed somewhat since first writing, indicating adoption trends that may suggest what their futures will be like. (Community size is an important predictor of the quality and amount of work that will go into improving the most-used open-source implementations of these languages; both growth and decline tend to be self-reinforcing.)

[128] Query for statistics [http://sourceforge.net/softwaremap/trove_list.php?form_cat=160].
[129] Query for statistics [http://freshmeat.net/browse/160/?topic_id=160].

Broadly speaking, C and C++ and Emacs Lisp have remained stable across the 1997-2003 time period, appealing to much the same constituencies in 2003 as they did in 1997. C has gained slowly at the expense of older conventional languages such as FORTRAN; C++, on the other hand, has lost some ground to Java.

Perl usage has grown respectably, but the language itself has been stagnant for some time. Perl's internals are notoriously grubby; it's been understood for years that the language's implementation needs to be rewritten from scratch, but an attempt in 1999 failed and another seems presently stalled in mid-2003. Nevertheless, Perl is still the 800-pound gorilla of scripting languages, and dominates Web scripting and CGI.

Tcl has been in a period of relative decline, or at least of diminishing visibility. In 1996 a widely-reported and plausible estimate of community sizes held that for every Python hacker there were five Tcl hackers and twelve Perl hackers. Today the SourceForge figures suggest those ratios are about 3:1:7. However, Tcl is reported to be very widely used for scripting of specialized components in several industries, including electronic design automation, radio and television broadcasting, and the film industry.

Python has risen in popularity as rapidly as Tcl has fallen. Though the Perl community is still twice the size of Python's, a visible tendency of the brightest Perl hackers to migrate to Python has been rather ominous for the former language — especially as there is no migration at all in the opposite direction.

Java has become widely used at sites already invested in Sun Microsystems technology and is in increasing deployment as an instructional language in undergraduate computer science curricula. Elsewhere, however, it is only marginally more popular than it was in 1997. Sun's determination to stick to a proprietary licensing model has prevented the major breakout many observers then predicted; under Linux and in the wider open-source community Java has not made the headway against C that it has elsewhere.

No new general-purpose language has emerged to seriously challenge those we've surveyed here. PHP is making inroads in Web development, challenging Perl CGIs (as well as ASP and server-side Java) but is almost never used for standalone programming. Non-Emacs Lisp dialects, a once-promising area that seemed headed for a renaissance in the mid-1990s, have continued to fade. Recent efforts such as Ruby (a sort of Python-Perl-Smalltalk cross developed in Japan) and Squeak (an open-source Smalltalk port) look promising, but have so far neither attracted hackers far outside their development groups nor demonstrated staying power.

Choosing an X Toolkit

An issue related to choice of language is choice of X toolkit for GUI programming. Recall the discussion in Chapter 1 of how X separates mechanism from policy. Each possible choice of toolkit will give you a slightly different look and feel.

Your choice of X toolkit will be connected to your choice of application language in two ways: first, because some languages ship with a binding to a preferred toolkit, and second because some toolkits only have bindings to a limited set of languages.

Java, of course, has its own cross-platform toolkits built in, so your choice will be between AWT (universally deployed) and Swing (more capable, more complex, slower, and only in JDK 1.2/Java 2). The remainder of this section focuses on the other languages we have surveyed. Similarly, if you're using Tcl, Tk comes bundled. There probably is not a lot of point in evaluating alternatives.

The once-ubiquitous Motif toolkit is effectively dead. It couldn't keep up with the newer toolkits distributed without license fees or restrictions. These attracted more developer effort until they surged past closed-source toolkits in capability and features; nowadays, the competition is all in open source.

The four toolkits to consider seriously in 2003 are Tk, GTK, Qt, and wxWindows, with GTK and Qt being the clear front runners. All four have ports on MacOS and Windows, so any choice will give you the capability to do cross-platform development.

The Tk toolkit is the oldest of the four and has the advantage of incumbency; it's native in Tcl and bindings to it are shipped with the stock version of Python. Libraries to provide language bindings to Tk are generally available for C and C++. Unfortunately, Tk also shows its age in that its standard widget set is both limited and rather ugly. On the other hand, the Tk Canvas widget has capabilities that other toolkits still match only with difficulty.

GTK began life as a replacement for Motif, and was invented to support the GIMP. It is now the preferred toolkit of the GNOME project and is used by hundreds of GNOME applications. The native API is C; bindings are available for C++, Perl, and Python, but do not ship with the stock language distributions. It's the only one of these four with a native C binding.

Qt is a toolkit associated with the KDE project. It is natively a C++ library; bindings are available for Python and Perl but do not ship with the stock interpreters. Qt has a reputation for having the best-designed and most expressive API of these four, but adoption was initially hindered by

controversy over early versions of the Qt license and was further slowed down by the fact that a C binding was slow in coming.

wxWindows is also natively C++ with bindings available in Perl and Python. The wxWindows developers emphasize their support for cross-platform development heavily and appear to regard it as the main selling point of the toolkit. Another selling point is that wxWindows is actually a wrapper around the native (GTK, Windows, and MacOS 9) widgets on each platform, so applications written using it retain a native look and feel.

As of mid-2003 few detailed comparisons have been written, but a Web search for “X toolkit comparison” may turn up some useful hits. Table 14.2 summarizes the state of play.

Table 14.2. Summary of X Toolkits.

Toolkit	**Native language**	**Shipped with**	**Bindings**				
C	**C++**	**Perl**	**Tcl**	**Python**			
Tk	Tcl	Tcl, Python	Y	Y	Y	Y	Y
GTK	C	Gnome	Y	Y	Y	Y	Y
Qt	C++	KDE	Y	Y	Y	Y	Y
wxWindows	C++	—	—	Y	Y	Y	Y

Architecturally, these libraries are all written at about the same abstraction level. GTK and Qt use a slot-and-signal apparatus for event-handling so similar that ports between them have been reported to be almost trivial. Your choice among them will probably be conditioned more by the availability of bindings to your chosen development language than anything else.

Chapter 15. Tools

The Tactics of Development

Unix is user-friendly — it's just choosy about who its friends are.
--
<author>Anonymous</author>

A Developer-Friendly Operating System

Unix has a long-established reputation as a good environment to develop under. It's well equipped with tools written by programmers for programmers. These automate away many of the grubby little tasks that would otherwise distract you from concentrating on the most important (and most enjoyable!) aspect of development— your design.

While all the tools you'll need are there and individually well documented, they're not knit together by an integrated development environment (IDE). Finding and assembling them into a kit that suits your needs has traditionally taken considerable effort.

If you're used to a good IDE — the kind of GUI-driven combination of editor, configuration-manager, compiler, and debugger now common on Macintosh and Windows systems — the Unix approach may seem casual, murky, and primitive. But there's actually method in it.

IDEs make a lot of sense for single-language programming in a tool-poor environment. If what you're doing is confined to grinding out C or C++ code by hand and the yard, they're quite appropriate. Under Unix, however, your languages and implementation options are a lot more varied. It's common to use multiple code generators, custom configurators, and many other standard and custom tools.

IDEs do exist under Unix (there are several good open-source ones, including emulations of the major Macintosh and Windows IDEs). But it's difficult to control an open-ended variety of programming tools with them, and they're not much used. Unix encourages a more flexible style, one less exclusively centered on the edit/compile/debug loop.

In this chapter we introduce you to the tactics of development under Unix — building code, managing code configurations, profiling, debugging, and automating away a lot of the drudgery associated with these tasks so you can concentrate on the fun parts. As usual, the exposition focuses

more on the architectural picture than the how-to details. When you *want* how-to details, most of the tools in this chapter are well described in *Programming with GNU Software* [Loukides-Oram].

Many of these tools automate things that you could do yourself by hand, albeit more slowly and with a higher error rate. The one-time cost of climbing the learning curve should be more than paid off by the ability to write programs more efficiently, and spend less attention on low-level details and more on design.

Unix programmers traditionally learn how to use these tools by osmosis from other programmers, and by exploration over a period of years. If you're a novice, pay careful attention; we're going to try to jump you over a big section of the Unix learning curve by showing you what is possible right at the outset. If you are an experienced Unix programmer in a hurry, you can skip this chapter — but maybe you shouldn't. There might just be some bit of useful lore here that even you don't know.

Choosing an Editor

The first and most basic tool of development is a text editor suitable for modifying and writing programs.

Literally dozens of text editors are available under Unix; writing one seems to be one of the standard finger exercises for budding open-source hackers. Most of these are ephemera, not suitable for extended use by anyone other than their authors. A few are emulations of non-Unix editors, useful as transition aids for programmers used to other operating systems. You can browse through a wide variety at SourceForge or ibiblio or any other major open-source archive.

For serious editing work, two editors completely dominate the Unix programming scene. Each is available in a couple of minor variant implementations, but has a standard version you can rely on finding on any modern Unix system. These two editors are *vi* and *Emacs*. We discussed them in Chapter 13 as part of our discussion of the right size of software.

As we noted in Chapter 13, these two editors express sharply contrasting design philosophies, but both are extremely popular and command great loyalty from identifiable core user populations. Surveys of Unix programmers consistently indicate about a 50/50 split between them, with all other editors barely registering.

In our earlier examinations of *vi* and *Emacs*, we were primarily concerned with their optional complexity and the surrounding design-philosophy issues. Many other things are worth knowing about these editors, both as a matter of practicality and of Unix cultural literacy.

Useful Things to Know about *vi*

The name of *vi* is an abbreviation for "visual editor" and is pronounced */vee eye/* (not */vie/* and *definitely* not */siks/*!).

vi was not quite the earliest screen-oriented editor; that palm goes to the Rand editor, *re*, that ran on Version 6 Unix in the 1970s. But *vi* is the longest-lived screen-oriented editor built for Unix that is still in use, and is a hallowed part of Unix tradition.

The original *vi* was the version present in the earliest BSD software distributions beginning in 1976; it is now obsolete. Its replacement was 'new vi' which shipped with 4.4BSD and is found on modern 4.4BSD variants such as BSD/OS, FreeBSD, and NetBSD systems. There are several variants with extended features, notably *vim*, *vile*, *elvis*, and *xvi*; of these *vim* is probably the most popular and is found on many Linux systems. All the variants are rather similar and share a core command set unchanged from the original *vi*.

Ports of *vi* are available for the Windows operating systems and MacOS.

Most introductory Unix books include a chapter describing basic *vi* usage. One place a *vi* FAQ is available is the Editor FAQ/vi [http://www.faqs.org/faqs/editor-faq/vi/]; you can find many other copies with a WWW keyword search for page titles including "vi" and "FAQ".

Useful Things to Know about Emacs

Emacs stands for 'EDiting MACroS' (pronounce it */ee´·maks/*). It was originally written in the late 1970s as a set of macros in an editor called TECO, then reimplemented several times in different ways. In an amusing twist, modern Emacs implementations include a TECO emulation mode.

In our earlier discussion of editors and optional complexity, we noted that many people consider Emacs excessively heavyweight. However, investing the time to learn it can yield rich rewards in productivity. Emacs supports many powerful editing modes that offer help with the syntax of various programming languages and markups. We'll see later in this chapter how Emacs can be used in combination with other development tools to give capabilities comparable to (and in many ways surpassing) those of conventional IDEs.

The standard Emacs, universally available on modern Unixes, is *GNU Emacs*; this is what generally runs if you type **emacs** to a Unix shell prompt. GNU Emacs sources and documentation are available at the Free Software Foundation archive site [ftp://gnu.org/pub/gnu].

The only major variant is called *XEmacs*; it has a better X interface but otherwise quite similar capabilities (it forked from Emacs 19). *XEmacs* has a home page [http://www.xemacs.org]. Emacs (and Emacs Lisp) is universally available under modern Unixes. It has been ported to MS-DOS (where it works poorly) and Windows 95 and NT (where it is said to work reasonably well).

Emacs includes its own interactive tutorial and very complete on-line documentation; you'll find instructions on how to invoke both on the default Emacs startup screen. A good introduction on paper is *Learning GNU Emacs* [Cameron].

The keystroke commands used in the Unix ports of Netscape/Mozilla and Internet Explorer text windows (in forms and the mailer) are copied from the stock Emacs bindings for basic text editing. These bindings are the closest thing to a cross-platform standard for editor keystrokes.

The Antireligious Choice: Using Both

Many people who regularly use both *vi* and Emacs tend to use them for different things, and find it valuable to know both.

In general, *vi* is best for small jobs — quick replies to mail, simple tweaks to system configuration, and the like. It is especially useful when you're using a new system (or a remote one over a network) and don't have your Emacs customization files handy.

Emacs comes into its own for extended editing sessions in which you have to handle complex tasks, modify multiple files, and use results from other programs during the session. For programmers using X on their console (which is typical on modern Unixes), it's normal to start up Emacs shortly after login time in a large window and leave it running forever, possibly visiting dozens of files and even running programs in multiple Emacs subwindows.

Special-Purpose Code Generators

Unix has a long-standing tradition of hosting tools that are specifically designed to generate code for various special purposes. The venerable monuments of this tradition, which go back to Version 7 and earlier days, and were actually used to write the original Portable C Compiler back in the 1970s, are lex(1) and yacc(1). Their modern, upward-compatible successors are flex(1) and bison(1), part of the GNU toolkit and still heavily used today. These programs have set an example that is carried forward in projects like GNOME's *Glade* interface builder.

yacc and *lex*

yacc and *lex* are tools for generating language parsers. We observed in Chapter 8 that your first minilanguage is all too likely to be an accident rather than a design. That accident is likely to have a hand-coded parser that costs you far too much maintenance and debugging time — especially if you have not realized it is a parser, and have thus failed to properly separate it from the remainder of your application code. Parser generators are tools for doing better than an accidental, ad-hoc implementation; they don't just let you express your grammar specification at a higher level, they also wall off all the parser's implementation complexity from the rest of your code.

If you reach a point where you are planning to implement a minilanguage from scratch, rather than by extending or embedding an existing scripting language or parsing XML, *yacc* and *lex* will probably be your most important tools after your C compiler.

lex and *yacc* each generate code for a single function — respectively, "get a token from the input stream" and "parse a sequence of tokens to see if it matches a grammar". Usually, the *yacc*-generated parser function calls a Lex-generated tokenizer function each time it wants to get another token. If there are no user-written C callbacks at all in the *yacc*-generated parser, all it will do is a syntax check; the value returned will tell the caller if the input matched the grammar it was expecting.

More usually, the user's C code, embedded in the generated parser, populates some runtime data structures as a side-effect of parsing the input. If the minilanguage is declarative, your application can use these runtime data structures directly. If your design was an imperative minilanguage, the data structures might include a parse tree which is immediately fed to some kind of evaluation function.

yacc has a rather ugly interface, through exported global variables with the name prefix `yy_`. This is because it predates structs in C; in fact, *yacc* predates C itself; the first implementation was written in C's predecessor B. The crude though effective algorithm *yacc*-generated parsers use to try to recover from parse errors (pop tokens until an explicit error production is matched) can also lead to problems, including memory leaks.

> If you are building parse trees, using malloc to make nodes, and you start popping things off the stack in error recovery, you don't get to recover (free) the storage. In general, Yacc can't do it, since it doesn't know enough about what's on the stack. If the yacc parser were in C++, it could assume that the values were classes and "destruct" them. In "real" compilers, parse tree nodes are generated

> using an arena-based allocator, so the nodes don't leak, but there is a logical leak anyway that needs to be thought about to make industrial-strength error recovery.
>
> —
>
> <author>SteveJohnson</author>

lex is a lexical analyzer generator. It's a member of the same functional family as grep(1) and awk(1), but more powerful because it enables you to arrange for arbitrary C code to be executed on each match. It accepts a declarative minilanguage and emits skeleton C code.

A crude but useful way to think about what a *lex*-generated tokenizer does is as a sort of inverse grep(1). Where grep(1) takes a single regular expression and returns a list of matches in the incoming data stream, each call to a *lex*-generated tokenizer takes a list of regular expressions and indicates which expression occurs next in the datastream.

> Splitting input analysis into tokenizing input and parsing the token stream is a useful tactic even if you're not using Yacc and Lex and your "tokens" are nothing like the usual ones in a compiler. More than once I've found that splitting input handling into two levels made the code much simpler and easier to understand, despite the complexity added by the split itself.
>
> —
>
> <author>HenrySpencer</author>

lex was written to automate the task of generating lexical analyzers (tokenizers) for compilers. It turned out to have a surprisingly wide range of uses for other kinds of pattern recognition, and has since been described as "the Swiss-army knife of Unix programming".[130]

If you are attacking any kind of pattern-recognition or state-machine problem in which all the possible input stimuli will fit in a byte, *lex* may enable you to generate code that will be more efficient and reliable than a hand-crafted state machine.

> John Jarvis at Holmdel [an AT&T laboratory] used *lex* to find faults in circuit boards, by scanning the board, using a chain-encoding technique to represent the edges of areas on the board, and then using Lex to define patterns that would catch common fabrication errors.
>
> —
>
> <author>MikeLesk</author>

[130] The common latter-day description of Perl as a "Swiss-army chainsaw" is derivative.

Most importantly, the *lex* specification minilanguage is much higher-level and more compact than equivalent handcrafted C. Modules are available to use *flex*, the open-source version, with Perl (find them with a Web search for “lex perl”), and a work-alike implementation is part of *PLY* in Python.

lex generates parsers that are up to an order of magnitude slower than hand-coded parsers. This is not a good reason to hand-code, however; it’s an argument for prototyping with *lex* and hand-hacking only if prototyping reveals an actual bottleneck.

yacc is a parser generator. It, too, was written to automate part of the job of writing compilers. It takes as input a grammar specification in a declarative minilanguage resembling BNF (Backus-Naur Form) with C code associated with each element of the grammar. It generates code for a parser function that, when called, accepts text matching the grammar from an input stream. As each grammar element is recognized, the parser function runs the associated C code.

The combination of *lex* and *yacc* is very effective for writing language interpreters of all kinds. Though most Unix programmers never get to do the kind of general-purpose compiler-building that these tools were meant to assist, they’re extremely useful for writing parsers for run-control file syntaxes and domain-specific minilanguages.

lex-generated tokenizers are very fast at recognizing low-level patterns in input streams, but the regular-expression minilanguage that *lex* knows is not good at counting things, or recognizing recursively nested structures. For parsing those, you want *yacc*. On the other hand, while you theoretically could write a *yacc* grammar to do its own token-gathering, the grammar to specify that would be hugely bloated and the parser extremely slow. For tokenizing input, you want *lex*. Thus, these tools are symbiotic.

If you can implement your parser in a higher-level language than C (which we recommend you do; see Chapter 14 for discussion), then look for equivalent facilities like Python’s PLY (which covers both *lex* and *yacc*)[131] or Perl’s PY and Parse::Yapp modules, or Java’s CUP,[132] Jack,[133] or Yacc/M[134] packages.

As with macro processors, one of the problems with code generators and preprocessors is that compile-time errors in the generated code may carry line numbers that are relative to the generated code (which you don’t want to edit) rather than the generator input (which is where you need to

[131] PLY is downloadable [http://systems.cs.uchicago.edu/ply/].
[132] CUP is downloadable [http://www.cs.princeton.edu/~appel/modern/java/CUP/].
[133] Jack is downloadable [http://www.javaworld.com/javaworld/jw-12-1996/jw-12-jack.html].
[134] Yacc/M is downloadable [http://david.tribble.com/yaccm.html].

make corrections). *yacc* and *lex* address this by generating the same `#line` constructs that the C preprocessor does; these set the current line number for error reporting so the numbers will come out right. Any program that generates C or C++ should do likewise.

More generally, well-designed procedural-code generators should never require the user to hand-alter or even look at the generated parts. Getting those right is the code generator's job.

Case Study: The `fetchmailrc` Grammar

The canonical demonstration example that seems to have appeared in every *lex* and *yacc* tutorial ever written is a toy interactive calculator program that parses and evaluates arithmetic expressions entered by the user. We will spare you yet another repetition of this cliche; if you are interested, consult the source code of the bc(1) and dc(1) calculator implementations from the GNU project, or the paradigm example 'hoc'[135] from [Kernighan-Pike84].

Instead, the grammar of *fetchmail*'s run-control-file parser provides a good medium-sized case study in *lex* and *yacc* usage. There are a couple of points of interest here.

The *lex* specification, in `rcfile_l.l`, is a very typical implementation of a shell-like syntax. Note how two complementary rules support either single or double-quoted strings; this is a good idea in general. The rules for accepting (possibly signed) integer literals and discarding comments are also pretty generic.

The *yacc* specification, in `rcfile_y.y`, is long but straightforward. It does not perform any *fetchmail* actions, just sets bits in a list of internal control blocks. After startup, *fetchmail*'s normal mode of operation is just to repeatedly walk that list, using each record to drive a retrieval session with a remote site.

Case Study: *Glade*

We looked at *Glade* in Chapter 8 as a good example of a declarative minilanguage. We also noted that its back end produces a result by generating code in any one of several languages.

Glade is a good modern example of an application-code generator. What makes it Unixy in spirit are the following features, which most GUI builders (especially most proprietary GUI builders) don't have:

[135] http://cm.bell-labs.com/cm/cs/upe/

- Rather than being glued together as one monster monolith, the *Glade* GUI and *Glade* code generator obey the Rule of Separation (following the “separated engine and interface” design pattern).
- The GUI and code generator are connected by an (XML-based) textual data file format that can be read and modified by other tools.
- Multiple target languages (as opposed to just C or C++) are supported. More could easily be added.

The design implies that it should also be possible to replace the *Glade* GUI editor component, should that ever become desirable.

make: Automating Your Recipes

Program sources by themselves don’t make an application. The way you put them together and package them for distribution matters, too. Unix provides a tool for semi-automating these processes; make(1). *Make* is covered in most introductory Unix books. For a really thorough reference, you can consult *Managing Projects with Make* [Oram-Talbot]. If you’re using *GNU make* (the most advanced make, and the one normally shipped with open-source Unixes) the treatment in *Programming with GNU Software* [Loukides-Oram] may be better in some respects. Most Unixes that carry *GNU make* will also support GNU Emacs; if yours does you will probably find a complete make manual on-line through Emacs’s *info* documentation system.

Ports of *GNU make* to DOS and Windows are available from the FSF.

Basic Theory of *make*

If you’re developing in C or C++, an important part of the recipe for building your application will be the collection of compilation and linkage commands needed to get from your sources to working binaries. Entering these commands is a lot of tedious detail work, and most modern development environments include a way to put them in command files or databases that can automatically be re-executed to build your application.

Unix’s make(1) program, the original of all these facilities, was designed specifically to help C programmers manage these recipes. It lets you write down the dependencies between files in a project in one or more ‘makefiles’. Each makefile consists of a series of *productions*; each one tells *make* that some given target file depends on some set of source files, and says what to do if any of

the sources are newer than the target. You don't actually have to write down all dependencies, as the *make* program can deduce a lot of the obvious ones from filenames and extensions.

For example: You might put in a makefile that the binary `myprog` depends on three object files `myprog.o`, `helper.o`, and `stuff.o`. If you have source files `myprog.c`, `helper.c`, and `stuff.c`, *make* will know without being told that each `.o` file depends on the corresponding `.c` file, and supply its own standard recipe for building a `.o` file from a `.c` file.

> Make originated with a visit from Steve Johnson (author of *yacc*, etc.), storming into my office, cursing the Fates that had caused him to waste a morning debugging a correct program (bug had been fixed, file hadn't been compiled, **cc *.o** was therefore unaffected). As I had spent a part of the previous evening coping with the same disaster on a project I was working on, the idea of a tool to solve it came up. It began with an elaborate idea of a dependency analyzer, boiled down to something much simpler, and turned into Make that weekend. Use of tools that were still wet was part of the culture. Makefiles were text files, not magically encoded binaries, because that was the Unix ethos: printable, debuggable, understandable stuff.
>
> —
>
> <author>StuartFeldman</author>

When you run **make** in a project directory, the *make* program looks at all productions and timestamps and does the minimum amount of work necessary to make sure derived files are up to date.

You can read a good example of a moderately complex makefile in the sources for *fetchmail*. In the subsections below we'll refer to it again.

Very complex makefiles, especially when they call subsidiary makefiles, can become a source of complications rather than simplifying the build process. A now-classic warning is issued in *Recursive Make Considered Harmful*.[136] The argument in this paper has become widely accepted since it was written in 1997, and has come near to reversing previous community practice.

No discussion of make(1) would be complete without an acknowledgement that it includes one of the worst design botches in the history of Unix. The use of tab characters as a required leader for command lines associated with a production means that the interpretation of a makefile can change drastically on the basis of invisible differences in whitespace.

[136] Available on the Web [http://www.tip.net.au/~millerp/rmch/recu-make-cons-harm.html].

> Why the tab in column 1? Yacc was new, Lex was brand new. I hadn't tried either, so I figured this would be a good excuse to learn. After getting myself snarled up with my first stab at Lex, I just did something simple with the pattern newline-tab. It worked, it stayed. And then a few weeks later I had a user population of about a dozen, most of them friends, and I didn't want to screw up my embedded base. The rest, sadly, is history.
>
> —
> <author>StuartFeldman</author>

make in Non-C/C++ Development

make is not just useful for C/C++ recipes, however. Scripting languages like those we described in Chapter 14 may not require conventional compilation and link steps, but there are often other kinds of dependencies that make(1) can help you with.

Suppose, for example, that you actually generate part of your code from a specification file, using one of the techniques from Chapter 9. You can use *make* to tie the spec file and the generated source together. This will ensure that whenever you change the spec and remake, the generated code will automatically be rebuilt.

It's quite common to use makefile productions to express recipes for making documentation as well as code. You'll often see this approach used to automatically generate PostScript or other derived documentation from masters written in some markup language (like HTML or one of the Unix document-macro languages we'll survey in Chapter 18). In fact, this sort of use is so common that it's worth illustrating with a case study.

Case Study: *make* for Document-File Translation

In the fetchmail makefile, for example, you'll see three productions that relate files named `FAQ`, `FEATURES`, and `NOTES` to HTML sources `fetchmail-FAQ.html`, `fetchmail-features.html`, and `design-notes.html`.

The HTML files are meant to be accessible on the fetchmail Web page, but all the HTML markup makes them uncomfortable to look at unless you're using a browser. So the `FAQ`, `FEATURES`, and `NOTES` are flat-text files meant to be flipped through quickly with an editor or pager program by someone reading the *fetchmail* sources themselves (or, perhaps, distributed to FTP sites that don't support Web access).

The flat-text forms can be made from their HTML masters by using the common open-source program lynx(1). *lynx* is a Web browser for text-only displays; but when invoked with the `-dump` option it functions reasonably well as an HTML-to-ASCII formatter.

With the productions in place, the developer can edit the HTML masters without having to remember to manually rebuild the flat-text forms afterwards, secure in the knowledge that `FAQ`, `FEATURES`, and `NOTES` will be properly rebuilt whenever they are needed.

Utility Productions

Some of the most heavily used productions in typical makefiles don't express file dependencies at all. They're ways to bundle up little procedures that a developer wants to mechanize, like making a distribution package or removing all object files in order to do a build from scratch.

> Non-file productions were intentional and in there from day one. 'Make all' and 'clean' were my own conventions from earliest days. One of the older Unix jokes is "Make love" which results in "Don't know how to make love".
>
> —
>
> <author>StuartFeldman</author>

There is a well-developed set of conventions about what utility productions should be present and how they should be named. Following these will make your makefile much easier to understand and use.

`all`
: Your `all` production should make every executable of your project. Usually the `all` production doesn't have an explicit rule; instead it refers to all of your project's top-level targets (and, not accidentally, documents what those are). Conventionally, this should be the first production in your makefile, so it will be the one executed when the developer types **make** with no argument.

`test` Run the program's automated test suite, typically consisting of a set of unit tests[137] to find regressions, bugs, or other deviations from expected behavior during the development process. The 'test' production can also be used by end-users of the software to ensure that their installation is functioning correctly.

`clean` Remove all files (such as binary executables and object files) that are normally created when you **make all**. A **make clean** should reset the process of building the software to a good initial state.

`dist` Make a source archive (usually with the tar(1) program) that can be shipped as a unit and used to rebuild the program on another machine. This target should do the equivalent of depending on **all** so that a **make dist** automatically rebuilds the whole project before making the distribution archive — this is a good way to avoid last-minute embarrassments, like not shipping derived files that are actually needed (like the flat-text `README` in *fetchmail*, which is actually generated from an HTML source).

`distclean` Throw away everything but what you would include if you were bundling up the source with **make dist**. This may be the the same as **make clean** but should be included as a production of its own anyway, to document what's going on. When it's different, it usually differs by throwing away local configuration files that aren't part of the normal **make all** build sequence (such as those generated by autoconf(1); we'll talk about autoconf(1) in Chapter 17).

`realclean` Throw away everything you can rebuild using the makefile. This may be the same as **make distclean**, but should be included as a production of its own anyway, to document what's going on. When it's different, it usually differs by throwing away files that are derived but (for whatever reason) shipped with the project sources anyway.

[137] A unit test is test code attached to a module to verify correct performance. Use of the term 'unit test' suggests that the test is written concurrently with the code by the developer of the code, and implies a discipline in which module releases aren't considered complete until they have attached test code. The term and the concept originated in the "Extreme Programming" methodology popularized by Kent Beck, but has gained wide acceptance among Unix programmers since about 2001.

`install` — Install the project's executables and documentation in system directories so they will be accessible to general users (this typically requires root privileges). Initialize or update any databases or libraries that the executables require in order to function.

`uninstall` — Remove files installed in system directories by **make install** (this typically requires root privileges). This should completely and perfectly reverse a **make install**. The presence of an uninstall production implies a kind of humility that experienced Unix hands look for as a sign of thoughtful design; conversely, not having an uninstall production is at best careless, and (when, for example, an installation creates large database files) can be quite rude and thoughtless.

Working examples of all the standard targets are available for inspection in the *fetchmail* makefile. By studying all of them together you will see a pattern emerge, and (not incidentally) learn much about the *fetchmail* package's structure. One of the benefits of using these standard productions is that they form an implicit roadmap of their project.

But you need not limit yourself to these utility productions. Once you master *make*, you'll find yourself more and more often using the makefile machinery to automate little tasks that depend on your project file state. Your makefile is a convenient central place to put these; using it makes them readily available for inspection and avoids cluttering up your workspace with trivial little scripts.

Generating Makefiles

One of the subtle advantages of Unix *make* over the dependency databases built into many IDEs is that makefiles are simple text files — files that can be generated by programs.

In the mid-1980s it was fairly common for large Unix program distributions to include elaborate custom shellscripts that would probe their environment and use the information they gathered to construct custom makefiles. These custom configurators reached absurd sizes. I wrote one once that was 3000 lines of shell, about twice as large as any single module in the program it was configuring — and this was not unusual.

The community eventually said "Enough!" and various people set out to write tools that would automate away part or all of the process of maintaining makefiles. These tools generally tried to address two issues:

One issue is *portability*. Makefile generators are commonly built to run on many different hardware platforms and Unix variants. They generally try to deduce things about the local system (including everything from machine word size up to which tools, languages, service libraries, and even document formatters it has available). They then try to use those deductions to write makefiles that exploit the local system's facilities and compensate for its quirks.

The other issue is *dependency derivation*. It's possible to deduce a great deal about the dependencies of a collection of C sources by analyzing the sources themselves (especially by looking at what include files they use and share). Many makefile generators do this in order to mechanically generate *make* dependencies.

Each different makefile generator tackles these objectives in a slightly different way. Probably a dozen or more generators have been attempted, but most proved inadequate or too difficult to drive or both, and only a few are still in live use. We'll survey the major ones here. All are available as open-source software on the Internet.

makedepend

Several small tools have tackled the rule automation part of the problem exclusively. This one, distributed along with the X windowing system from MIT, is the fastest and most useful and comes preinstalled under all modern Unixes, including all Linuxes.

makedepend takes a collection of C sources and generates dependencies for the corresponding `.o` files from their `#include` directives. These can be appended directly to a makefile, and in fact makedepend is defined to do exactly that.

makedepend is useless for anything but C projects. It doesn't try to solve more than one piece of the makefile-generation problem. But what it does it does quite well.

makedepend is sufficiently documented by its manual page. If you type **man makedepend** at a terminal window you will quickly learn what you need to know about invoking it.

Imake

Imake was written in an attempt to mechanize makefile generation for the X window system. It builds on *makedepend* to tackle both the dependency-derivation and portability problems.

Imake system effectively replaces conventional makefiles with Imakefiles. These are written in a more compact and powerful notation which is (effectively) compiled into makefiles. The

compilation uses a rules file which is system-specific and includes a lot of information about the local environment.

Imake is well suited to X's particular portability and configuration challenges and universally used in projects that are part of the X distribution. However, it has not achieved much popularity outside the X developer community. It's hard to learn, hard to use, hard to extend, and produces generated makefiles of mind-numbing size and complexity.

The *Imake* tools will be available on any Unix that supports X, including Linux. There has been one heroic effort [DuBois] to make the mysteries of *Imake* comprehensible to non-X-programming mortals. These are worth learning if you are going to do X programming.

autoconf

autoconf was written by people who had seen and rejected the *Imake* approach. It generates per-project `configure` shellscripts that are like the old-fashioned custom script configurators. These `configure` scripts can generate makefiles (among other things).

Autoconf is focused on portability and does no built-in dependency derivation at all. Although it is probably as complex as *Imake*, it is much more flexible and easier to extend. Rather than relying on a per-system database of rules, it generates `configure` shell code that goes out and searches your system for things.

Each `configure` shellscript is built from a per-project template that you have to write, called `configure.in`. Once generated, though, the `configure` script will be self-contained and can configure your project on systems that don't carry autoconf(1) itself.

The *autoconf* approach to makefile generation is like imake's in that you start by writing a makefile template for your project. But *autoconf*'s `Makefile.in` files are basically just makefiles with placeholders in them for simple text substitution; there's no second notation to learn. If you want dependency derivation, you must take explicit steps to call makedepend(1) or some similar tool — or use automake(1).

autoconf is documented by an on-line manual in the GNU *info* format. The source scripts of autoconf are available from the FSF archive site, but are also preinstalled on many Unix and Linux versions. You should be able to browse this manual through your Emacs's help system.

Despite its lack of direct support for dependency derivation, and despite its generally ad-hoc approach, in mid-2003 *autoconf* is clearly the most popular of the makefile generators, and has

been for some years. It has eclipsed *Imake* and driven at least one major competitor (*metaconfig*) out of use.

A reference, *GNU Autoconf, Automake and Libtool* is available [Vaughan]. We'll have more to say about *autoconf*, from a slightly different angle, in Chapter 17.

automake

automake is an attempt to add *Imake*-like dependency derivation as a layer on top of autoconf(1). You write `Makefile.am` templates in a broadly *Imake*-like notation; automake(1) compiles them to `Makefile.in` files, which *autoconf*'s `configure` scripts then operate on.

automake is still relatively new technology in mid-2003. It is used in several FSF projects but has not yet been widely adopted elsewhere. While its general approach looks promising, it is as yet rather brittle — it works when used in stereotyped ways but tends to break badly if you try to do anything unusual with it.

Complete on-line documentation is shipped with *automake*, which can be downloaded from the FSF archive site.

Version-Control Systems

Code evolves. As a project moves from first-cut prototype to deliverable, it goes through multiple cycles in which you explore new ground, debug, and then stabilize what you've accomplished. And this evolution doesn't stop when you first deliver for production. Most projects will need to be maintained and enhanced past the 1.0 stage, and will be released multiple times. Tracking all that detail is just the sort of thing computers are good at and humans are not.

Why Version Control?

Code evolution raises several practical problems that can be major sources of friction and drudgery — thus a serious drain on productivity. Every moment spent on these problems is a moment not spent on getting the design and function of your project right.

Perhaps the most important problem is *reversion*. If you make a change, and discover it's not viable, how can you revert to a code version that is known good? If reversion is difficult or unreliable, it's hard to risk making changes at all (you could trash the whole project, or make many hours of painful work for yourself).

Almost as important is *change tracking*. You know your code has changed; do you know why? It's easy to forget the reasons for changes and step on them later. If you have collaborators on a project, how do you know what they have changed while you weren't looking, and who was responsible for each change?

> Amazingly often, it is useful to ask what *you* have changed since the last known-good version, even if you have no collaborators. This often uncovers unwanted changes, such as forgotten debugging code. I now do this routinely before checking in a set of changes.
>
> —
>
> <author>HenrySpencer</author>

Another issue is *bug tracking*. It's quite common to get new bug reports for a particular version after the code has mutated away from it considerably. Sometimes you can recognize immediately that the bug has already been stomped, but often you can't. Suppose it doesn't reproduce under the new version. How do you get back the state of the code for the old version in order to reproduce and understand it?

To address these problems, you need procedures for keeping a history of your project, and annotating it with comments that explain the history. If your project has more than one developer, you also need mechanisms for making sure developers don't overwrite each others' versions.

Version Control by Hand

The most primitive (but still very common) method is all hand-hacking. You snapshot the project periodically by manually copying everything in it to a backup. You include history comments in source files. You make verbal or email arrangements with other developers to keep their hands off certain files while you hack them.

The hidden costs of this hand-hacking method are high, especially when (as frequently happens) it breaks down. The procedures take time and concentration; they're prone to error, and tend to get slipped under pressure or when the project is in trouble — that is, exactly when they are most needed.

As with most hand-hacking, this method does not scale well. It restricts the granularity of change tracking, and tends to lose metadata details such as the order of changes, who did them, and why. Reverting just a part of a large change can be tedious and time consuming, and often developers are forced to back up farther than they'd like after trying something that doesn't work.

Automated Version Control

To avoid these problems, you can use a *version-control system* (VCS), a suite of programs that automates away most of the drudgery involved in keeping an annotated history of your project and avoiding modification conflicts.

Most VCSs share the same basic logic. To use one, you start by *registering* a collection of source files — that is, telling your VCS to start archive files describing their change histories. Thereafter, when you want to edit one of these files, you have to *check out* the file — assert an exclusive lock on it. When you're done, you *check in* the file, adding your changes to the archive, releasing the lock, and entering a change comment explaining what you did.

The history of the project is not necessarily linear. All VCSs in common use actually allow you to maintain a tree of variant versions (for ports to different machines, say) with tools for merging branches back into the main "trunk" version. This feature becomes important as the size and dispersion of the development group increases. It needs to be used with care, however; multiple active variants of the code base can be very confusing (just associated bug reports to the right version are not necessarily easy), and automated merging of branches does not guaranteed that the combined code works.

Most of the rest of what a VCS does is convenience: labeling, and reporting features surrounding these basic operations, and tools which allow you to view differences between versions, or to group a given set of versions of files as a named *release* that can be examined or reverted to at any time without losing later changes.

VCSs have their problems. The biggest one is that using a VCS involves extra steps every time you want to edit a file, steps that developers in a hurry tend to want to skip if they have to be done by hand. Near the end of this chapter we'll discuss a way to solve this problem.

Another problem is that some kinds of natural operations tend to confuse VCSs. Renaming files is a notorious trouble spot; it's not easy to automatically ensure that a file's version history will be carried along with it when it is renamed. Renaming problems are particularly difficult to resolve when the VCS supports branching.

Despite these difficulties, VCSs are a huge boon to productivity and code quality in many ways, even for small single-developer projects. They automate away many procedures that are just tedious work. They help a lot in recovering from mistakes. Perhaps most importantly, they free programmers to experiment by guaranteeing that reversion to a known-good state will always be easy.

(VCSs, by the way, are not merely good for program code; the manuscript of this book was maintained as a collection of files under *RCS* while it was being written.)

Unix Tools for Version Control

Historically, three VCSs have been of major significance in the Unix world, and we'll survey them here. For an extended introduction and tutorial, consult *Applying RCS and SCCS* [Bolinger-Bronson].

Source Code Control System (SCCS)

The first was *SCCS*, the original Source Code Control System developed by Bell Labs around 1980 and featured in System III Unix. *SCCS* seems to have been the first serious attempt at a unified source-code management system; concepts that it pioneered are still found at some level in all later ones, including commercial Unix and Windows products such as ClearCase.

SCCS itself is, however, now obsolete; it was proprietary Bell Labs software. Superior open-source alternatives have since been developed, and most of the Unix world has converted to those. *SCCS* is still in use to manage old projects at some commercial vendors, but can no longer be recommended for new projects.

No complete open-source implementation of *SCCS* exists. A clone called CSSC (Compatibly Stupid Source Control) is in development under the sponsorship of the FSF.

Revision Control System (RCS)

The superior open-source alternatives began with RCS (Revision Control System), born at Purdue University a few years after *SCCS* and originally distributed with 4.3BSD Unix. It is logically similar to *SCCS* but has a cleaner command interface, and good facilities for grouping together entire project releases under symbolic names.

RCS is currently the most widely used version control system in the Unix world. Some other Unix version-control systems use it as a back end or underlayer. It is well suited for single-developer or small-group projects hosted at a single development shop.

The *RCS* sources are maintained and distributed by the FSF. Free ports are available for Microsoft operating systems and VAX VMS.

Concurrent Version System (CVS)

CVS (Concurrent Version System) began life as a front end to *RCS* developed in the early 1990s, but the model of version control it uses was different enough that it immediately qualified as a new design. Modern implementations don't rely on *RCS*.

Unlike *RCS* and *SCCS*, *CVS* doesn't exclusively lock files when they're checked out. Instead, it tries to reconcile nonconflicting changes mechanically when they're checked back in, and requests human help on conflicts. The design works because patch conflicts are much less common than one might intuitively think.

The interface of *CVS* is significantly more complex than that of *RCS*, and it needs a lot more disk space. These properties make it a poor choice for small projects. On the other hand, *CVS* is well suited to large multideveloper efforts distributed across several development sites connected by the Internet. *CVS* tools on a client machine can easily be told to direct their operations to a repository located on a different host.

The open-source community makes heavy use of *CVS* for projects such as GNOME and Mozilla. Typically, such *CVS* repositories allow anyone to check out sources remotely. Anyone can, therefore, make a local copy of a project, modify it, and mail change patches to the project maintainers. Actual write access to the repository is more limited and has to be explicitly granted by the project maintainers. A developer who has such access can perform a commit option from his modified local copy, which will cause the local changes to get made directly to the remote repository.

You can see an example of a well-run *CVS* repository, accessible over the Internet, at the GNOME CVS site [http://cvs.gnome.org]. This site illustrates the use of *CVS*-aware browsing tools such as Bonsai, which are useful in helping a large and decentralized group of developers coordinate their work.

The social machinery and philosophy accompanying the use of *CVS* is as important as the details of the tools. The assumption is that projects *will* be open and decentralized, with code subject to peer review and inspection even by developers who are not officially members of the project group.

Just as importantly, *CVS*'s nonlocking philosophy means that projects can't be blocked by a lock if a programmer disappears in the middle of making some changes. *CVS* thus allows developers to avoid the “single person point of failure” problem; in turn, this means that project boundaries can be fluid, casual contributions are relatively easy, and projects are not required to have an elaborate hierarchy of control.

The *CVS* sources are maintained and distributed by the FSF.

CVS has significant problems. Some are merely implementation bugs, but one basic problem is that your project's file namespace is not versioned in the same way changes to files themselves are. Thus, *CVS* is easily confused by file renamings, deletions, and additions. Also, *CVS* records changes on a per-file basis, rather than as *sets* of changes made to files. This makes it harder to back out to specific versions, and harder to handle partial check-ins. Fortunately, none of these problems are intrinsic to the nonlocking style, and they have been successfully addressed by newer version-control systems.

Other Version-Control Systems

CVS's design problems are sufficient to have created demand for a better open-source VCS. Several such efforts are under way as of 2003. The most notable of these are *Aegis* and *Subversion*.

Aegis [http://www.pcug.org.au/~millerp/aegis/aegis.html] has the longest history of any of these alternatives, has hosted its own development since 1991, and is a mature production system. It features a heavy emphasis on regression-testing and validation.

Subversion [http://subversion.tigris.org/] is positioned as "CVS done right", with the known design problems fully addressed, and in 2003 probably has the best near-term prospect of replacing *CVS*.

The BitKeeper [http://www.bitkeeper.com] project explores some interesting design ideas related to change-sets and multiple distributed code repositories. Linus Torvalds uses Bitkeeper for the Linux kernel sources. Its non-open-source license is, however, controversial, and has significantly retarded the acceptance of the product.

Runtime Debugging

Anyone who has been programming longer than a week knows that getting the syntax of your programming language right is the *easy* part of debugging. The hard part comes after that, when you need to understand why your syntactically correct program doesn't behave as you expect.

The Unix tradition encourages developers to anticipate this problem by designing for transparency — in particular, designing programs in such a way that their internal data flows are readily monitored with the naked eye and simple tools, and readily mentally modeled. This is a topic we covered in detail in Chapter 6. Design for transparency is valuable both for preventing bugs and for easing the runtime-debugging task.

Design for transparency is not, however, sufficient in itself. When you are debugging a program at runtime, it's extremely useful to be able to examine the state of your program at runtime, set breakpoints, and execute pieces of it down to the single-statement level in a controlled way. Unix has a long tradition of hosting programs to help you with this. Open-source Unixes feature a powerful one called *gdb* (yet another FSF project) that supports C and C++ debugging.

Perl, Python, Java, and Emacs Lisp all support standard packages or programs (included with their base distributions) that allow you to set breakpoints, control execution, and do general runtime-debugger things. Tcl, designed as a small language for small projects, has no such facility (though it does have a trace facility that can be used to watch variables at runtime).

Remember the Unix philosophy. Spend your time on design quality, not the low-level details, and automate away everything you can — including the detail work of runtime debugging.

Profiling

As a general rule, 90% of the execution time of your program will be spent in 10% of its code. Profilers are tools that help you identify the 10% of hot spots that constrain the speed of your program. This is a good thing for making it faster.

But in the Unix tradition, profilers have a far more important function. They enable you *not* to optimize the other 90%! This is good, and not just because it saves you work. The *really* valuable effect is that not optimizing that 90% holds down global complexity and reduces bugs.

You may recall that we quoted Donald Knuth observing “Premature optimization is the root of all evil” in Chapter 1, and that Rob Pike and Ken Thompson had a few pungent observations on the topic as well. These were the voices of experience. Do good design. Think about what's *right* first. Tune for efficiency later.

Profilers help you do this. If you get in the good habit of using them, you can get rid of the bad habit of premature optimization. Profilers don't just change the way you work; they change how you think.

Profilers for compiled languages rely on instrumenting object code, so they are even more platform-dependent than compilers. On the other hand, a compiled-language profiler doesn't care about the source language of the programs it instruments. Under Unix, the single profiler gprof(1) handles C, C++, and all other compiled languages.

Perl, Python, and Emacs Lisp have their own profilers included in their basic distributions; these are portable across all platforms on which the host languages themselves run. Java has built-in profiling. Tcl has no profiling support as yet.

Combining Tools with Emacs

One of the things the Emacs editor is very good at is acting as a front end for other development tools (we discussed this from a philosophical angle in Chapter 13). In fact, nearly every tool we've discussed in this chapter can be driven from within an Emacs editor session through front ends that give them greater utility than they would have running standalone.

To illustrate this, we'll walk you through the use of these tools with Emacs in a typical build/test/debug cycle. For details on them, see Emacs's own on-line help system; this section just gives you an overview, to motivate you to learn more.

Read and learn — not just about Emacs, but about the mental habit of looking for synergies between programs, and creating them. Try to read this section as instruction in philosophy, not just technique.

Emacs and *make*

Make, for example, can be started with the Emacs command **ESC-x compile** followed by an Enter. This command will run make(1) in the current directory, capturing the output in an Emacs buffer.

This by itself wouldn't be very useful. But Emacs's *make* mode knows about the error message format (featuring a source file and line number) emitted by Unix C compilers and many other tools.

If anything run by *make* issues error messages, the command **Ctl-X ‘** (control-X-backquote) will try to parse them and take you to each error location in turn, popping open a window on the appropriate file and taking the cursor to the error line.[138]

This makes it extremely easy to step through an entire build, fixing any syntax that has been broken since the last compile.

Emacs and Runtime Debugging

[138] Look at p+processes->compile under the Emacs help menu for more information on these and related compilation-control commands.

For catching runtime errors, Emacs offers similar integration with your symbolic debugger — that is, you can use an Emacs mode to set breakpoints in your programs and examine their runtime state. You run the debugger by sending it commands through an Emacs window. Whenever the debugger stops on a breakpoint, the message the debugger ships back about the source location is parsed and used to pop up a window on the source around the breakpoint.

Emacs's Grand Unified Debugger mode supports all the major C debuggers: gdb(1), sdb(1), dbx(1), and xdb(1). It also supports Perl symbolic debugging using the perldb module, and the standard debuggers for both Java and Python. Facilities built into Emacs Lisp itself support interactive debugging of Emacs Lisp code.

At time of writing (mid-2003) there is not yet support for Tcl debugging from within Emacs. The design of Tcl is such that it seems unlikely to be added.

Emacs and Version Control

Once you've corrected your program's syntax and fixed its runtime bugs, you may want to save the changes into a version-controlled archive. If you've only tried running version-control tools from the shell, it's hard to blame you for sloughing off this important step. Who wants to have to remember to run checkout/checkin commands around every edit operation?

Fortunately, Emacs offers help here too. Code built into Emacs implements a simple-to-use front end for *SCCS*, *RCS*, *CVS*, or Subversion. The single command **Ctl-x v v** tries to deduce the next logical version-control operation to do on the file you are visiting. The operations this includes are registering a file, checking out and locking it, and checking it back in (accepting a change comment in a pop-up buffer).[139]

Emacs also helps you view the change history of version-controlled files, and helps you back out changes you don't want. It makes it easy to apply version-control operations to whole sets or project directory trees of files. In general, it does a pretty good job of making version-control operations painless.

The implications of these features are larger than you might guess before you've gotten used to it. You'll find, once you get used to fast and easy version control, that it's extremely liberating. Because you know you can always revert to a known-good state, you'll find you feel more free to develop in a fluid and exploratory way, trying lots of changes out to see their effects.

[139] See the subsection of the Emacs on-line documentation titled *Version Control* for more details on these and related commands.

Emacs and Profiling

Surprise...this is perhaps the only phase of the development cycle in which Emacs front-ending does *not* offer substantial help. Profiling is an intrinsically batchy operation — instrument your program, run it, view the statistics, speed-tune the code with an editor, repeat. There isn't much room for Emacs leverage in the profiling-specific parts of this cycle.

Nevertheless, there's a good tutorial reason for us to think about Emacs and profiling. If you found yourself analyzing a *lot* of profiling reports, it might pay you to write a mode in which a mouse click or keystroke on a profile report line visited the source of the relevant function. This actually would be fairly easy to do using the Emacs 'tags' code. In fact, by the time you read this, some other reader may already have written such a mode and contributed it to the public Emacs code base.

The real point here is again a philosophical one. Don't drudge — drudging wastes your time and productivity! If you find yourself spending a lot of time on the low-level mechanical parts of development, step back. Apply the Unix philosophy. Use your toolkit to automate or semi-automate the task.

Then give back something in return for all you've inherited, by posting your solution as open-source software to the Internet. Help liberate your fellow programmers from drudgery, too.

Like an IDE, Only Better

Earlier in this chapter we asserted that Emacs can give you capabilities resembling those of a conventional integrated development environment, only better. By now you should have enough facts in hand to see how that can be true. You can run entire development projects from inside Emacs, driving the low-level mechanics with a few keystrokes and saving yourself the mental effort and disruption of constantly switching contexts.

The Emacs-enabled development style trades away some capabilities of advanced IDEs, like graphical views of program structure. But those are frills. What Emacs gives you in return is flexibility and control. You're not limited by the imagination of the IDE designer: you can tweak, customize, and add task-related intelligence using Emacs Lisp. Also, Emacs is better at supporting mixed-language development than conventional IDEs.

Finally, you're not limited to accepting what one small group of IDE developers sees fit to support. By keeping an eye on the open-source community, you can benefit from the work of thousands of your peers, Emacs-using developers facing challenges much like yours. This is much more effective — and much more fun.

Chapter 16. Reuse

On Not Reinventing the Wheel

When the superior man refrains from acting, his force is felt for a thousand miles.
-- *Tao Te Ching (as popularly mistranslated)*

Reluctance to do unnecessary work is a great virtue in programmers. If the Chinese sage Lao-Tze were alive today and still teaching the way of the Tao, he would probably be mistranslated as: When the superior programmer refrains from coding, his force is felt for a thousand miles. In fact, recent translators have suggested that the Chinese term *wu-wei* that has traditionally been rendered as “inaction” or “refraining from action” should probably be read as “least action” or “most efficient action” or “action in accordance with natural law”, which is an even better description of good engineering practice!

Remember the Rule of Economy. Re-inventing fire and the wheel for every new project is terribly wasteful. Thinking time is precious and very valuable relative to all the other inputs that go into software development; accordingly, it should be spent solving new problems rather than rehashing old ones for which known solutions already exist. This attitude gives the best return both in the “soft” terms of developing human capital and in the “hard” terms of economic return on development investment.

> Reinventing the wheel is bad not only because it wastes time, but because reinvented wheels are often square. There is an almost irresistible temptation to economize on reinvention time by taking a shortcut to a crude and poorly-thought-out version, which in the long run often turns out to be false economy.
>
> —
>
> <author>HenrySpencer</author>

The most effective way to avoid reinventing the wheel is to borrow someone else’s design and implementation of it. In other words, to reuse code.

Unix supports reuse at every level from individual library modules up to entire programs, which Unix helps you script and recombine. Systematic reuse is one of the most important distinguishing behaviors of Unix programmers, and the experience of using Unix should teach you a habit of trying to prototype solutions by combining existing components with a minimum of new invention, rather than rushing to write standalone code that will only be used once.

The virtuousness of code reuse is one of the great apple-pie-and-motherhood verities of software development. But many developers entering the Unix community from a basis of experience in other operating systems have never learned (or have unlearned) the habit of systematic reuse. Waste and duplicative work is rife, even though it seems to be against the interests both of those who pay for code and those who produce it. Understanding why such dysfunctional behavior persists is the first step toward changing it.

The Tale of J. Random Newbie

Why do programmers reinvent wheels? There are many reasons, reaching all the way from the narrowly technical to the psychology of programmers and the economics of the software production system. The damage from the endemic waste of programming time reaches all these levels as well.

Consider the first, formative job experience of J. Random Newbie, a programmer fresh out of college. Let us assume that he (or she) has been taught the value of code reuse and is brimming with youthful zeal to apply it.

Newbie's first project puts him on a team building some large application. Let's say for the sake of example that it's a GUI intended to help end users intelligently construct queries for and navigate through a large database. The project managers have assembled what they deem to be a suitable collection of tools and components, including not merely a development language but many libraries as well.

The libraries are crucial to the project. They package many services — from windowing widgets and network connections on up to entire subsystems like interactive help — that would otherwise require immense quantities of additional coding, with a severe impact on the project's budget and its ship date.

Newbie is a little worried about that ship date. He may lack experience, but he's read *Dilbert* and heard a few war stories from experienced programmers. He knows management has a tendency to what one might euphemistically call "aggressive" schedules. Perhaps he has read Ed Yourdon's *Death March* [Yourdon], which as long ago as 1996 noted that a majority of projects are on a time and resource budget at least 50% too tight, and that the trend is for that squeeze to get worse.

But Newbie is bright and energetic. He figures his best chance of succeeding is to learn to use the tools and libraries that have been handed to him as intelligently as possible. He limbers up his typing fingers, hurls himself at the challenge...and enters hell.

Everything takes longer and is more painful than he expects. Beneath the surface gloss of their demo applications, the components he is re-using seem to have edge cases in which they behave unpredictably or destructively — edge cases his code tickles daily. He often finds himself wondering what the library programmers were thinking. He can't tell, because the components are inadequately documented — often by technical writers who aren't programmers and don't think like programmers. And he can't read the source code to learn what it is actually doing, because the libraries are opaque blocks of object code under proprietary licenses.

Newbie has to code increasingly elaborate workarounds for component problems, to the point where the net gain from using the libraries starts to look marginal. The workarounds make his code progressively grubbier. He probably hits a few places where a library simply cannot be made to do something crucially important that is theoretically within its specifications. Sometimes he is sure there is some way to actually make the black box perform, but he can't figure out what it is.

Newbie finds that as he puts more strain on the libraries, his debugging time rises exponentially. His code is bedeviled with crashes and memory leaks that have trace paths leading into the libraries, into code he can't see or modify. He knows most of those trace paths probably lead back out to his code, but without source it is very difficult to trace through the bits he didn't write.

Newbie is growing horribly frustrated. He had heard in college that in industry, a hundred lines of finished code a week is considered good performance. He had laughed then, because he was many times more productive than that on his class projects and the code he wrote for fun. Now it's not funny any more. He is wrestling not merely with his own inexperience but with a cascade of problems created by the carelessness or incompetence of others — problems he can't fix, but can only work around.

The project schedule is slipping. Newbie, who dreamed of being an architect, finds himself a bricklayer trying to build with bricks that won't stack properly and that crumble under load-bearing pressure. But his managers don't want to hear excuses from a novice programmer; complaining too loudly about the poor quality of the components is likely to get him in political trouble with the senior people and managers who selected them. And even if he could win that battle, changing components would be a complicated proposition involving batteries of lawyers peering narrowly at licensing terms.

Unless Newbie is very, very lucky, he is not going to be able to get library bugs fixed within the lifetime of his project. In his saner moments, he may realize that the working code in the libraries doesn't draw his attention the way the bugs and omissions do. He'd love to sit down for a clarifying chat with the component developers; he suspects they can't be the idiots their code sometimes

suggests, just programmers like him working within a system that frustrates their attempts to do the right thing. But he can't even find out who they are — and if he could, the software vendor they work for probably wouldn't let them talk to him.

In desperation, Newbie starts making his own bricks — simulating less stable library services with more stable ones and writing his own implementations from scratch. His replacement code, because he has a complete mental model of it that he can refresh by rereading, tends to work relatively well and be easier to debug than the combination of opaque components and workarounds it replaces.

Newbie is learning a lesson; the less he relies on other peoples' code, the more lines of code he can get written. This lesson feeds his ego. Like all young programmers, deep down he thinks he is smarter than anyone else. His experience seems, superficially, to be confirming this. He begins building his own personal toolkit, one better fitted to his hand.

Unfortunately, the roll-your-own reflexes Newbie is acquiring are a short-term local optimization that will cause long-term problems. He may get more lines of code written, but the actual value of what he produces is likely to drop substantially relative to what it would have been if he were doing successful reuse. More code does not equal better code, not when it's written at a lower level and largely devoted to reinventing wheels.

Newbie has at least one more demoralizing experience in store, when he changes jobs. He is likely to discover that he can't take his toolkit with him. If he walks out of the building with code he wrote on company time, his old employers could well regard this as intellectual-property theft. His new employers, knowing this, are not likely to react well if he admits to reusing any of his old code.

Newbie could well find his toolkit is useless even if he can sneak it into the building at his new job. His new employers may use a different set of proprietary tools, languages, and libraries. It is likely he will have to learn a somewhat new set of techniques and reinvent a new set of wheels each time he changes projects.

Thus do programmers have reuse (and other good practices that go with it, like modularity and transparency) systematically conditioned out of them by a combination of technical problems, intellectual-property barriers, politics, and personal ego needs. Multiply J. Random Newbie by a hundred thousand, age him by decades, and have him grow more cynical and more used to the system year by year. There you have the state of much of the software industry, a recipe for enormous waste of time and capital and human skill — even *before* you factor in vendors' market-control tactics, incompetent management, impossible deadlines, and all the other pressures that make doing good work difficult.

The professional culture that springs from J. Random Newbie's experiences will reflect them in the large. Programming shops will have a ferocious Not Invented Here complex. They will be poisonously ambivalent about code reuse, pushing inadequate but heavily marketed vendor components on their programmers in order to meet schedule crunches, while simultaneously rejecting reuse of the programmers' own tested code. They will churn out huge volumes of ad-hoc, duplicative software produced by programmers who know the results will be garbage but are glumly resigned to never being able to fix anything but their own individual pieces.

The closest equivalent of code reuse to emerge in such a culture will be a dogma that code once paid for can never be thrown away, but must instead be patched and kluged even when all parties know that it would be better to scrap and start anew. The products of this culture will become progressively more bloated and buggy over time even when every individual involved is trying his or her hardest to do good work.

Transparency as the Key to Reuse

We field-tested the tale of J. Random Newbie on a number of experienced programmers. If you the reader are one yourself, we expect you responded to it much as they did: with groans of recognition. If you are not a programmer but you manage programmers, we sincerely hope you found it enlightening. The tale is intended to illustrate the ways in which different levels of pressure against reuse reinforce each other to create a magnitude of problem not linearly predictable from any individual cause.

So accustomed are most of us to the background assumptions of the software industry that it can take considerable mental effort before the primary causes of this problem can be separated from the accidents of narrative. But they are not, in the end, very complex.

At the bottom of most of J. Random Newbie's troubles (and the large-scale quality problems they imply) is transparency — or, rather, the lack of it. You can't fix what you can't see inside. In fact, for any software with a nontrivial API, you can't even properly *use* what you can't see inside. Documentation is inadequate not merely in practice but in principle; it cannot convey all the nuances that the code embodies.

In Chapter 6, we observed how central transparency is to good software. Object-code-only components destroy the transparency of a software system, On the other hand, the frustrations of code reuse are far less likely to bite when the code you are attempting to reuse is available for reading and modification. Well-commented source code is its own documentation. Bugs in source

code can be fixed. Source can be instrumented and compiled for debugging to make probing its behavior in obscure cases easier. And if you need to change its behavior, you can do that.

There is another vital reason to demand source code. A lesson Unix programmers have learned through decades of constant change is that source code lasts, object code doesn't. Hardware platforms change, service components like support libraries change, the operating system grows new APIs and deprecates old ones. Everything changes — but opaque binary executables cannot adapt to change. They are brittle, cannot be reliably forward-ported, and have to be supported with increasingly thick and error-prone layers of emulation code. They lock users into the assumptions of the people who built them. You need source because, even if you have neither the intention nor the need to change the software, you will have to rebuild it in new environments to keep it running.

The importance of transparency and the code-legacy problem are reasons that you should require the code you reuse to be open to inspection and modification.[140] It is not a complete argument for what is now called 'open source'; because 'open source' has rather stronger implications than simply requiring code to be transparent and visible.

From Reuse to Open Source

In the early days of Unix, components of the operating system, its libraries, and its associated utilities were passed around as source code; this openness was a vital part of the Unix culture. We described in Chapter 2 how, when this tradition was disrupted after 1984, Unix lost its initial momentum. We have also described how, a decade later, the rise of the GNU toolkit and Linux prompted a rediscovery of the value of open-source code.

Today, open-source code is again one of the most powerful tools in any Unix programmer's kit. Accordingly, though the explicit concept of "open source" and the most widely used open-source licenses are decades younger than Unix itself, it's important to understand both to do leading-edge development in today's Unix culture.

Open source relates to code reuse in much the way romantic love relates to sexual reproduction — it's possible to explain the former in terms of the latter, but to do so is to risk overlooking much of what makes the former interesting. Open source does not reduce to merely being a tactic for supporting reuse in software development. It is an emergent phenomenon, a social contract among

[140]NASA, which consciously builds software intended to have a service life of decades, has learned to insist on source-code availability for all space avionics software.

developers and users that tries to secure several advantages related to transparency. As such, there are several different ways to approaching an understanding of it.

Our historical description earlier in this book chose one angle by focusing on causal and cultural relationships between Unix and open source. We'll discuss the institutions and tactics of open-source development in Chapter 19. In discussing the theory and practice of code reuse, it's useful to think of open source more specifically, as a direct response to the problems we dramatized in the tale of J. Random Newbie.

Software developers want the code they use to be transparent. Furthermore, they don't want to lose their toolkits and their expertise when they change jobs. They get tired of being victims, fed up with being frustrated by blunt tools and intellectual-property fences and having to repeatedly re-invent the wheel.

These are the motives for open source that flow from J. Random Newbie's painful initiatory experience with reuse. Ego needs play a part here, too; they give pervasive emotional force to what would otherwise be a bloodless argument about engineering best practices. Software developers are like every other kind of craftsman and artificer; they want, not so secretly, to be artists. They have the drives and needs of artists, including the desire to have an audience. They not only want to reuse code, they want their code to be reused. There is an imperative here that goes beyond and overrides short-term economic goal-seeking and that cannot be satisfied by closed-source software production.

Open source is a kind of ideological preemptive strike on all these problems. If the root of most of J. Random Newbie's problems with reuse is the opacity of closed-source code, then the institutional assumptions that produce closed-source code must be smashed. If corporate territoriality is a problem, it must be attacked or bypassed until the corporations have caught on to how self-destructive their territorial reflexes are. Open source is what happens when code reuse gets a flag and an army.

Accordingly, since the late 1990s, it no longer makes any sense to try to recommend strategies and tactics for code reuse without talking about open source, open-source practices, open-source licensing, and the open-source community. Even if those issues could be separated elsewhere, they have become inextricably bound together in the Unix world.

In the remainder of this chapter, we'll survey various issues associated with re-using open-source code: evaluation, documentation, and licensing. In Chapter 19 we'll discuss the open-source

development model more generally, and examine the conventions you should follow when you are releasing code for others to use.

The Best Things in Life Are Open

On the Internet, literally terabytes of Unix sources for systems and applications software, service libraries, GUI toolkits and hardware drivers are available for the taking. You can have most built and running in minutes with standard tools. The mantra is **./configure; make; make install**; usually you have to be root to do the install part.

People from outside the Unix world (especially non-technical people) are prone to think open-source (or ‘free’) software is necessarily inferior to the commercial kind, that it’s shoddily made and unreliable and will cause more headaches than it saves. They miss an important point: in general, open-source software is written by people who care about it, need it, use it themselves, and are putting their individual reputations among their peers on the line by publishing it. They also tend to have less of their time consumed by meetings, retroactive design changes, and bureaucratic overhead. They are therefore both more strongly motivated and better positioned to do excellent work than wage slaves toiling Dilbert-like to meet impossible deadlines in the cubicles of proprietary software houses.

Furthermore, the open-source user community (those peers) is not shy about nailing bugs, and its standards are high. Authors who put out substandard work experience a lot of social pressure to fix their code or withdraw it, and can get a lot of skilled help fixing it if they choose. As a result, mature open-source packages are generally of high quality and often functionally superior to any proprietary equivalent. They may lack polish and have documentation that assumes much, but the vital parts will usually work quite well.

Besides the peer-review effect, another reason to expect better quality is this: in the open-source world developers are never forced by a deadline to close their eyes, hold their noses, and ship. A major consequent difference between open-source practice and elsewhere is that a release level of 1.0 actually means the software is ready to use. In fact, a version number of 0.90 or above is a fairly reliable signal that the code is production-ready, but the developers are not quite ready to bet their reputations on it.

If you are a programmer from outside the Unix world, you may find this claim difficult to believe. If so, consider this: on modern Unixes, the C compiler itself is almost invariably open source. The Free Software Foundation’s GNU Compiler Collection (GCC) is so powerful, so well documented, and so

reliable that there is effectively no proprietary Unix compiler market left, and it has become normal for Unix vendors to port GCC to their platforms rather than do in-house compiler development.

The way to evaluate an open-source package is to read its documentation and skim some of its code. If what you see appears to be competently written and documented with care, be encouraged. If there also is evidence that the package has been around for a while and has incorporated substantial user feedback, you may bet that it is quite reliable (but test anyway).

A good gauge of maturity and the volume of user feedback is the number of people besides the original author mentioned in the `README` and project news or history files in the source distribution. Credits to lots of people for sending in fixes and patches are signs both of a significant user base keeping the authors on their toes, and of a conscientious maintainer who is responsive to feedback and will take corrections. It is also an indication that, if early code tends to be a minefield of bugs, there has since been a thundering herd run through it without too many recent explosions.

It's also a good omen when the software has its own Web page, on-line FAQ (Frequently Asked Questions) list, and an associated mailing list or Usenet newsgroup. These are all signs that a live and substantial community of interest has grown up around the software. On Web pages, recent updates and an extensive mirror list are reliable signs of a project with a vigorous user community. Packages that are duds just don't get this kind of continuing investment, because they can't reward it.

Ports to multiple platforms are also a valuable indication of a diversified user base. Project pages tend to advertise new ports precisely because they signal credibility.

Here are some examples of what Web pages associated with high-quality open-source software look like:

- GIMP [http://www.gimp.org/]
- GNOME [http://www.gnome.org]
- KDE [http://www.kde.org]
- Python [http://www.python.org]
- The Linux kernel [http://www.kernel.org]

- PostgreSQL [http://www.postgresql.org]
- XFree86 [http://xfree86.org]
- InfoZip [http://www.info-zip.org/pub/infozip/]

Looking at Linux distributions is another good way to find quality. Distribution-makers for Linux and other open-source Unixes carry a lot of specialist expertise about which projects are best-of-breed — that's a large part of the value they add when they integrate a release. If you are already using an open-source Unix, something else to check is whether the package you are evaluating is already carried by your distribution.

Where to Look?

Because so much open source is available in the Unix world, skill at finding code to reuse can have an enormous payoff — much greater than is the case for other operating systems. Such code comes in many forms: individual code snippets and examples, code libraries, utilities to be reused in scripts. Under Unix most code reuse is not a matter of actual cut-and-paste into your program — in fact, if you find yourself doing that, there is almost certainly a more graceful mode of reuse that you are missing. Accordingly, one of the most useful skills to cultivate under Unix is a good grasp of all the different ways to glue together code, so you can use the Rule of Composition.

To find re-usable code, start by looking under your nose. Unixes have always featured a rich toolkit of re-usable utilities and libraries; modern ones, such as any current Linux system, include thousands of programs, scripts, and libraries that may be re-usable. A simple **man -k** search with a few keywords often yields useful results.

To begin to grasp something of the amazing wealth of resources out there, surf to SourceForge, ibiblio, and Freshmeat.net. Other sites as important as these three may exist by the time you read this book, but all three of these have shown continuing value and popularity over a period of years, and seem likely to endure.

SourceForge [http://www.sourceforge.net] is a demonstration site for software specifically designed to support collaborative development, complete with associated project-management services. It is not merely an archive but a free development-hosting service, and in mid-2003 is undoubtedly the largest single hub of open-source activity in the world.

The Linux archives at ibiblio [http://www.ibiblio.org] were the largest in the world before SourceForge. The ibiblio archives are passive, simply a place to publish packages. They do, however, have a better interface to the World Wide Web than most passive sites (the program that creates its Web look and feel was one of our case studies in the discussion of Perl in Chapter 14). It's also the home site of the Linux Documentation Project, which maintains many documents that are excellent resources for Unix users and developers.

Freshmeat [http://www.freshmeat.net] is a system dedicated to providing release announcements of new software, and new releases of old software. It lets users and third parties attach reviews to releases.

These three general-purpose sites contain code in many languages, but most of their content is C or C++. There are also sites specialized around some of the interpreted languages as discussed in Chapter 14.

The CPAN archive is the central repository for useful free code in Perl. It is easily reached from the Perl home page [http://www.perl.com/perl].

The Python Software Activity makes an archive of Python software and documentation available at the Python Home Page [http://www.python.org].

Many Java applets and pointers to other sites featuring free Java software are made available at the Java Applets page [http://java.sun.com/applets/].

One of the most valuable ways you can invest your time as a Unix developer is to spend time wandering around these sites learning what is available for you to use. The coding time you save may be your own!

Browsing the package metadata is a good idea, but don't stop there. Sample the code, too. You'll get a better grasp on what the code is doing, and be able to use it more effectively.

More generally, reading code is an investment in the future. You'll learn from it — new techniques, new ways to partition problems, different styles and approaches. Both using the code and learning from it are valuable rewards. Even if you don't use the techniques in the code you study, the improved definition of the problem you get from looking at other peoples' solutions may well help you invent a better one of your own.

Read before you write; develop the habit of reading code. There are seldom any completely new problems, so it is almost always possible to discover code that is close enough to what you need to

be a good starting point. Even when your problem is genuinely novel, it is likely to be genetically related to a problem someone else has solved before, so the solution you need to develop is likely to be related to some pre-existing one as well.

Issues in Using Open-Source Software

There are three major issues in using or re-using open-source software; quality, documentation, and licensing terms. We've seen above that if you exercise a little judgment in picking through your alternatives, you will generally find one or more of quite respectable quality.

Documentation is often a more serious issue. Many high-quality open-source packages are less useful than they technically ought to be because they are poorly documentated. Unix tradition encourages a rather hieratic style of documentation, one which (while it may technically capture all of a package's features) assumes that the reader is intimately familiar with the application domain and reading very carefully. There are good reasons for this, which we'll discuss in Chapter 18, but the style can present a bit of a barrier. Fortunately, extracting value from it is a learnable skill.

It is worth doing a Web search for phrases including the software package, or topic keywords, and the string “HOWTO” or “FAQ”. These queries will often turn up documentation more useful to novices than the man page.

The most serious issue in reusing open-source software (especially in any kind of commercial product) is understanding what obligations, if any, the package's license puts upon you. In the next two sections we'll discuss this issue in detail.

Licensing Issues

Anything that is not public domain has a copyright, possibly more than one. Under U.S. federal law, the authors of a work hold copyright even if there is no copyright notice.

Who counts as an author under copyright law can be complicated, especially for software that has been worked on by many hands. This is why licenses are important. They can authorize uses of code in ways that would be otherwise impermissible under copyright law and, drafted appropriately, can protect users from arbitrary actions by the copyright holders.

In the proprietary software world, the license terms are designed to protect the copyright. They're a way of granting a few rights to users while reserving as much legal territory as possible for the owner

(the copyright holder). The copyright holder is very important, and the license logic so restrictive that the exact technicalities of the license terms are usually unimportant.

As will be seen below, the copyright holder typically uses the copyright to protect the license, which makes the code freely available under terms he intends to perpetuate indefinitely. Otherwise, only a few rights are reserved and most choices pass to the user. In particular, the copyright holder cannot change the terms on a copy you already have. Therefore, in open-source software the copyright holder is almost irrelevant — but the license terms are very important.

Normally the copyright holder of a project is the current project leader or sponsoring organization. Transfer of the project to a new leader is often signaled by changing the copyright holder. However, this is not a hard and fast rule; many open-source projects have multiple copyright holders, and there is no instance on record of this leading to legal problems. Some projects choose to assign copyright to the Free Software Foundation, on the theory that it has an interest in defending open source and lawyers available to do it.

What Qualifies as Open Source

For licensing purposes, we can distinguish several different kinds of rights that a license may convey. There are rights to copy and redistribute, rights to use, rights to modify for personal use, and rights to redistribute modified copies. A license may restrict or attach conditions to any of these rights.

The Open Source Definition [http://www.opensource.org/osd.html] is the result of a great deal of thought about what makes software “open source” or (in older terminology) “free”. It is widely accepted in the open-source community as an articulation of the social contract among open-source developers. Its constraints on licensing impose the following requirements:

- An unlimited right to copy be granted.
- An unlimited right to redistribute in unmodified form be granted.
- An unlimited right to modify for personal use be granted.

The guidelines prohibit restrictions on redistribution of modified binaries; this meets the needs of software distributors, who need to be able to ship working code without encumbrance. It allows authors to require that modified sources be redistributed as pristine sources plus patches, thus establishing the author's intentions and an "audit trail" of any changes by others.

The OSD is the legal definition of the "OSI Certified Open Source" certification mark, and as good a definition of "free software" as anyone has ever come up with. All of the standard licenses (MIT, BSD, Artistic, GPL/LGPL, and MPL) meet it (though some, like GPL, have other restrictions which you should understand before choosing it).

Note that licenses that allow only noncommercial use do not qualify as open-source licenses, even if they are based on GPL or some other standard license. Such licenses discriminate against particular occupations, persons, and groups, a practice which the OSD's Clause 5 explicitly forbids.

Clause 5 was written after years of painful experience. No-commercial-use licenses turn out to have the problem that there is no bright-line legal test for what sort of redistribution qualifies as 'commercial'. Selling the software as a product qualifies, certainly. But what if it were distributed at a nominal price of zero in conjunction with other software or data, and a price is charged for the whole collection? Would it make a difference whether the software were essential to the function of the whole collection?

Nobody knows. The very fact that no-commercial-use licenses create uncertainty about a redistributor's legal exposure is a serious strike against them. One of the objectives of the OSD is to ensure that people in the distribution chain of OSD-conforming software do not need to consult with intellectual-property lawyers to know what their rights are. OSD forbids complicated restrictions against persons, groups, and occupations partly so that people dealing with collections of software will not face a combinatorial explosion of slightly differing (and perhaps conflicting) restrictions on what they can do with it.

This concern is not hypothetical, either. One important part of the open-source distribution chain is CD-ROM distributors who aggregate it in useful collections ranging from simple anthology CDs up to bootable operating systems. Restrictions that would make life prohibitively complicated for CD-ROM distributors, or others trying to spread open-source software commercially, have to be forbidden.

On the other hand, the OSD has nothing to say about the laws of your jurisdiction. Some countries have laws against exporting certain restricted technologies to named 'rogue states'. The OSD cannot negate those, it only says that licensors may not add restrictions of their own.

Standard Open-Source Licenses

Here are the standard open-source license terms you are likely to encounter. The abbreviations listed here are in general use.

MIT [http://www.opensource.org/licenses/mit-license.html]
: MIT X Consortium license (like BSD's but with no advertising requirement)

BSD [http://www.opensource.org/licenses/bsd-license.html]
: University of California at Berkeley Regents copyright (used on BSD code)

Artistic License [http://www.opensource.org/licenses/artistic-license.html]
: Same terms as Perl Artistic License

GPL [http://www.gnu.org/copyleft.html]
: GNU General Public License

LGPL [http://www.gnu.org/copyleft.html]
: Library (or 'Lesser') GPL

MPL [http://www.opensource.org/licenses/MPL-1.1.html]
: Mozilla Public License

We'll discuss these licenses in more detail, from a developer's point of view, in Chapter 19. For the purposes of this chapter, the only important distinction among them is whether they are infectious or not. A license is *infectious* if it requires that any derivative work of the licensed software also be placed under its terms.

Under these licenses, the only kind of open-source use you should really worry about is actual incorporation of the free-software code into a proprietary product (as opposed, say, to merely using open-source development tools to make your product). If you're prepared to include proper license acknowledgements and pointers to the source code you're using in your product documentation, even direct incorporation should be safe provided the license is not infectious.

The GPL is both the most widely used and the most controversial infectious license. And it is clause 2(b), requiring that any derivative work of a GPLed program itself be GPLed, that causes the controversy. (Clause 3(b) requiring licensors to make source available on physical media on demand used to cause some, but the Internet explosion has made publishing source code archives as required by 3(a) so cheap that nobody worries about the source-publication requirement any more.)

Nobody is quite certain what the “contains or is derived from” in clause 2(b) means, nor what kinds of use are protected by the “mere aggregation” language a few paragraphs later. Contentious issues include library linking and inclusion of GPL-licensed header files. Part of the problem is that the U.S. copyright statutes do not define what derivation is; it has been left to the courts to hammer out definitions in case law, and computer software is an area in which this process (as of mid-2003) has barely begun.

At one end, the “mere aggregation” certainly makes it safe to ship GPLed software on the same media with your proprietary code, provided they do not link to or call each other. They may even be tools operating on the same file formats or on-disk structures; that situation, under copyright law, would not make one a derivative of the other.

At the other end, splicing GPLed code into your proprietary code, or linking GPLed object code to yours, certainly does make your code a derivative work and requires it to be GPLed.

It is generally believed that one program may execute a second program as a subprocess without either program becoming thereby a derivative work of the other.

The case that causes dispute is dynamic linking of shared libraries. The Free Software Foundation’s position is that if a program calls another program as a shared library, then that program is a derivative work of the library. Some programmers think this claim is overreaching. There are technical, legal, and political arguments on both sides that we won’t rehash here. Since the Free Software Foundation wrote and owns the license, it would be prudent to behave as if the FSF’s position is correct until a court rules otherwise.

Some people think the 2(b) language is deliberately designed to infect every part of any commercial program that uses even a snippet of GPLed code; such people refer to it as the GPV, or “General Public Virus”. Others think the “mere aggregation” language covers everything short of mixing GPL and non-GPL code in the same compilation or linkage unit.

This uncertainty has caused enough agitation in the open-source community that the FSF had to develop the special, slightly more relaxed “Library GPL” (which they have since renamed the “Lesser GPL”) to reassure people they could continue to use runtime libraries that came with the FSF’s GNU compiler collection.

You’ll have to choose your own interpretation of clause 2(b); most lawyers will not understand the technical issues involved, and there is no case law. As a matter of empirical fact, the FSF has never (from its founding in 1984 to mid-2003, at least) sued anyone under the GPL but it has enforced

the GPL by threatening lawsuit, in all known cases successfully. And, as another empirical fact, Netscape includes the source and object of a GPLed program with the commercial distribution of its Netscape Navigator browser.

The MPL and LGPL are infectious in a more limited way than GPL. They explicitly allow linking with proprietary code without turning that code into a derivative work, provided all traffic between the GPLed and non-GPLed code goes through a library API or other well-defined interface.

When You Need a Lawyer

This section is directed to commercial developers considering incorporating software that falls under one of these standard licenses into closed-source products.

Having gone through all this legal verbiage, the expected thing for us to do at this point is to utter a somber disclaimer to the effect that we are not lawyers, and that if you have any doubts about the legality of something you want to do with open-source software, you should immediately consult a lawyer.

With all due respect to the legal profession, this would be fearful nonsense. The language of these licenses is as clear as legalese gets — they were written to be clear — and should not be at all hard to understand if you read it carefully. The lawyers and courts are actually more confused than you are. The law of software rights is murky, and case law on open-source licenses is (as of mid-2003) nonexistent; no one has ever been sued under them.

This means a lawyer is unlikely to have a significantly better insight than a careful lay reader. But lawyers are professionally paranoid about anything they don't understand. So if you ask one, he is rather likely to tell you that you shouldn't go anywhere near open-source software, despite the fact that he probably doesn't understand the technical aspects or the author's intentions anywhere near as well as you do.

Finally, the people who put their work under open-source licenses are generally not mega-corporations attended by schools of lawyers looking for blood in the water; they're individuals or volunteer groups who mainly want to give their software away. The few exceptions (that is, large companies both issuing under open-source licenses and with money to hire lawyers) have a stake in open source and don't want to antagonize the developer community that produces it by stirring up legal trouble. Therefore, your odds of getting hauled into court on an innocent technical violation are probably lower than your chances of being struck by lightning in the next week.

This isn't to say you should treat these licenses as jokes. That would be disrespectful of the creativity and sweat that went into the software, and you wouldn't enjoy being the first litigation target of an enraged author no matter how the lawsuit came out. But in the absence of definitive case law, a visible good-faith effort to meet the author's intentions is 99% of what you can do; the additional 1% of protection you might (or might not) get by consulting a lawyer is unlikely to make a difference.

Part IV. Community

Chapter 17. Portability

Software Portability and Keeping Up Standards

The realization that the operating systems of the target machines were as great an obstacle to portability as their hardware architecture led us to a seemingly radical suggestion: to evade that part of the problem altogether by moving the operating system itself.

--

<authorgroup>
<author>SteveJohnson</author>
<author>DennisRitchie</author>
</authorgroup>

Portability of C Programs and the UNIX System (1978)

Unix was the first production operating system to be ported between differing processor families (Version 6 Unix, 1976-77). Today, Unix is routinely ported to every new machine powerful enough to sport a memory-management unit. Unix applications are routinely moved between Unixes running on wildly differing hardware; in fact, it is unheard of for a port to fail.

Portability has always been one of Unix's principal advantages. Unix programmers tend to write on the assumption that hardware is evanescent and only the Unix API is stable, making as few assumptions as possible about machine specifics such as word length, endianness or memory architecture. In fact, code that is hardware-dependent in any way that goes beyond the abstract machine model of C is considered bad form in Unix circles, and only really tolerated in very special cases like operating system kernels.

Unix programmers have learned that it is easy to be wrong when anticipating that a software project will have a short lifetime.[141] Thus, they tend to avoid making software dependent on specific and perishable technologies, and to lean heavily on open standards. These habits of writing for portability are so ingrained in the Unix tradition that they are applied even to small single-use projects that are thought of as throwaway code. They have had secondary effects all through the design of the Unix development toolkit, and on programming languages like Perl and Python and Tcl that were developed under Unix.

[141] PDP-7 Unix and Linux were both examples of unexpected persistence. Unix originated as a research toy hacked together by some researchers between projects, half to play with file-system ideas and half to host a game. The second was summed up by its creator as "My terminal emulator grew legs" [Torvalds].

The direct benefit of portability is that it is normal for Unix software to outlive its original hardware platform, so tools and applications don't have to be reinvented every few years. Today, applications originally written for Version 7 Unix (1979) are routinely used not merely on Unixes genetically descended from V7, but on variants like Linux in which the operating system API was written from a Unix specification and shares no code with the Bell Labs source tree.

The indirect benefits are less obvious but may be more important. The discipline of portability tends to exert a simplifying influence on architectures, interfaces, and implementations. This both increases the odds of project success and reduces life-cycle maintenance costs.

In this chapter, we'll survey the scope and history of Unix standards. We'll discuss which ones are still relevant today and describe the areas of greater and lesser variance in the Unix API. We'll examine the tools and practices that Unix developers use to keep code portable, and develop some guides to good practice.

Evolution of C

The central fact of the Unix programming experience has always been the stability of the C language and the handful of service interfaces that always travel with it (notably, the standard I/O library and friends). The fact that a language originated in 1973 has required as little change as this one has in thirty years of heavy use is truly remarkable, and without parallels anywhere else in computer science or engineering.

In Chapter 4, we argued that C has been successful because it acts as a layer of thin glue over computer hardware approximating the "standard architecture" of [BlaauwBrooks]. There is, of course, more to the story than that. To understand the rest of the story, we'll need to take a brief look at the history of C.

Early History of C

C began life in 1971 as a systems-programming language for the PDP-11 port of Unix, based on Ken Thompson's earlier B interpreter which had in turn been modeled on BCPL, the Basic Common Programming Language designed at Cambridge University in 1966-67.[142]

[142]The 'C' in C therefore stands for Common — or, perhaps, for 'Christopher'. BCPL originally stood for "Bootstrap CPL"—a much simplified version of CPL, the very interesting but overly ambitious and never implemented Common Programming Language of Oxford and Cambridge, also known affectionately as "Christopher's Programming Language" after its prime advocate, computer-science pioneer Christopher Strachey.

Dennis Ritchie's original C compiler (often called the "DMR" compiler after his initials) served the rapidly growing community around Unix versions 5, 6, and 7. Version 6 C spawned Whitesmiths C, a reimplementation that became the first commercial C compiler and the nucleus of IDRIS, the first Unix workalike. But most modern C implementations are patterned on Steven C. Johnson's "portable C compiler" (PCC) which debuted in Version 7 and replaced the DMR compiler entirely in both System V and the BSD 4.x releases.

In 1976, Version 6 C introduced the `typedef`, `union`, and `unsigned int` declarations. The approved syntax for variable initializations and some compound operators also changed.

The original description of C was Brian Kernighan and Dennis M. Ritchie's original *The C Programming Language* aka "the White Book" [Kernighan-Ritchie]. It was published in 1978, the same year the Whitemiths C compiler became available.

The White Book described enhanced Version 6 C, with one significant exception involving the handling of public storage. Ritchie's original intention had been to model C's rules on FORTRAN COMMON declarations, on the theory that any machine that could handle FORTRAN would be ready for C. In the common-block model, a public variable may be declared multiple times; identical declarations are merged by the linker. But two early C ports (to Honeywell and IBM 360 mainframes) happened to be to machines with very limited common storage or a primitive linker or both. Thus, the Version 6 C compiler was moved to the stricter definition-reference model (requiring at most one definition of any given public variable and the **extern** keyword tagging references to it) described in [Kernighan-Ritchie].

This decision was reversed in the C compiler that shipped with Version 7 after it developed that a great deal of existing source depended on the looser rules. Pressure for backward-compatibility would foil yet another attempt to switch (in 1983's System V Release 1) before the ANSI Draft Standard finally settled on definition-reference rules in 1988. Common-block public storage is still admitted as an acceptable variation by the standard.

V7 C introduced `enum` and treated `struct` and `union` values as first-class objects that could be assigned, passed as arguments, and returned from functions (rather than being passed around by address).

> Another major change in V7 was that Unix data structure declarations were now documented on header files, and included. Previous Unixes had actually printed the data structures (e.g., for directories) in the manual, from which people would copy it into their code. Needless to say, this was a major portability problem.

—
<author>SteveJohnson</author>

The System III C version of the PCC compiler (which also shipped with BSD 4.1c) changed the handling of struct declarations so that members with the same names in different structs would not clash. It also introduced `void` and `unsigned char` declarations. The scope of `extern` declarations local to a function was restricted to the function, and no longer included all code following it.

The ANSI C Draft Proposed Standard added `const` (for read-only storage) and `volatile` (for locations such as memory-mapped I/O registers that might be modified asynchronously from the thread of program control). The `unsigned` type modifier was generalized to apply to any type, and a symmetrical `signed` was added. Initialization syntax for `auto` array and structure initializers and `union` types was added. Most importantly, function prototypes were added.

The most important changes in early C were the switch to definition-reference and the introduction of function prototypes in the Draft Proposed ANSI C Standard. The language has been essentially stable since copies of the X3J11 committee's working papers on the Draft Proposed Standard signaled the committee's intentions to compiler implementers in 1985-1986.

A more detailed history of early C, written by its designer, can be found at [Ritchie93].

C Standards

C standards development has been a conservative process with great care taken to preserve the spirit of the original C language, and an emphasis on ratifying experiments in existing compilers rather than inventing new features. The C9X charter[143] document is an excellent expression of this mission.

Work on the first official C standard began in 1983 under the auspices of the X3J11 ANSI committee. The major functional additions to the language were settled by the end of 1986, at which point it became common for programmers to distinguish between "K&R C" and "ANSI C".

> Many people don't realize how *unusual* the C standardization effort, especially the original ANSI C work, was in its insistence on standardizing only tested features. Most language standard committees spend much of their time inventing new features, often with little consideration of how they might be implemented. Indeed, the few ANSI C features that *were* invented from scratch — e.g., the

[143] Available on the Web [http://anubis.dkuug.dk/JTC1/SC22/WG14/www/charter].

> notorious “trigraphs”—were the most disliked and least successful features of C89.
>
> —
>
> <author>HenrySpencer</author>

> Void pointers were invented as part of the standards effort, and have been a winner. But Henry’s point is still well taken.
>
> —
>
> <author>SteveJohnson</author>

While the core of ANSI C was settled early, arguments over the contents of the standard libraries dragged on for years. The formal standard was not issued until the end of 1989, well after most compilers had implemented the 1985 recommendations. The standard was originally known as ANSI X3.159, but was redesignated ISO/IEC 9899:1990 when the International Standards Organization (ISO) took over sponsorship in 1990. The language variant it describes is generally known as C89 or C90.

The first book on C and Unix portability practice, *Portable C and Unix Systems Programming* [Lapin], was published in 1987 (I wrote it under a corporate pseudonym forced on me by my employers at the time). The Second Edition of [Kernighan-Ritchie] came out in 1988.

A very minor revision of C89, known as Amendment 1, AM1, or C93, was floated in 1993. It added more support for wide characters and Unicode. This became ISO/IEC 9899-1:1994.

Revision of the C89 standard began in 1993. In 1999, ISO/IEC 9899 (generally known as C99) was adopted by ISO. It incorporated Amendment 1, and added a great many minor features. Perhaps the most significant one for most programmers is the C++-like ability to declare variables at any point in a block, rather than just at the beginning. Macros with a variable number of arguments were also added.

The C9X working group has a Web page [http://anubis.dkuug.dk/JTC1/SC22/WG14/www/projects], but no third standards effort is planned as of mid-2003. They are developing an addendum on C for embedded systems.

Standardization of C has been greatly aided by the fact that working and largely compatible implementations were running on a wide variety of systems before standards work was begun. This made it harder to argue about what features should be in the standard.

Unix Standards

The 1973 rewrite of Unix in C made it unprecedentedly easy to port and modify. As a result, the ancestral Unix diverged into a family of operating systems early on. Unix standards originally developed to reconcile the APIs of the different branches of the family tree.

The Unix standards that evolved after 1985 were quite successful at this — so much so that they serve as valuable documentation of the API of modern Unix implementations. In fact, real-world Unixes follow published standards so closely that developers can (and frequently do) lean more on documents like the POSIX specification than on the official manual pages for the Unix variant they happen to be using.

In fact, on the newer open-source Unixes (such as Linux), it is common for operating-system features to have been engineered using published standards as the specification. We'll return to this point when we examine the RFC standards process later in this chapter.

Standards and the Unix Wars

The original motivation for the development of Unix standards was the split between the AT&T and Berkeley lines of development that we examined in Chapter 2.

The 4.x BSD Unixes were descended from the 1979 Version 7. After the release of 4.1BSD in 1980 the BSD line quickly developed a reputation as the cutting edge of Unix. Important additions included the *vi* visual editor, job control facilities for managing multiple foreground and background tasks from a single console, and improvements in signals (see Chapter 7). By far the most important addition was to be TCP/IP networking, but though Berkeley got the contract to do it in 1980, TCP/IP was not to ship in an external release for three years.

But another version, 1981's System III, became the basis of AT&T's later development. System III reworked the Version 7 terminals interface into a cleaner and more elegant form that was completely incompatible with the Berkeley enhancements. It retained the older (non-resetting) semantics of signals (again, see Chapter 7 for discussion of this point). The January 1983 release of System V Release 1 incorporated some BSD utilities (such as vi(1)).

The first attempt to bridge the gap came in February 1983 from UniForum, an influential Unix user group. Their Uniforum 1983 Draft Standard (UDS 83) described a “core Unix System” consisting of a subset of the System III kernel and libraries plus a file-locking primitive. AT&T declared support for UDS 83, but the standard was an inadequate subset of evolving practice based on

4.1BSD. The problem was exacerbated by the July 1983 release of 4.2BSD, which added many new features (including TCP/IP networking) and introduced some subtle incompatibilities with the ancestral Version 7.

The 1984 divestiture of the Bell operating companies and the beginnings of the Unix wars (see Chapter 2) significantly complicated matters. Sun Microsystems was leading the workstation industry in a BSD direction; AT&T was trying to get into the computer business and use control of Unix as a strategic weapon even as it continued to license the operating system to competitors like Sun. All the vendors were making business decisions to differentiate their versions of Unix for competitive advantage.

During the Unix wars, technical standardization became something that cooperating technical people pushed for and most product managers accepted grudgingly or actively resisted. The one large and important exception was AT&T, which declared its intention to cooperate with user groups in setting standards when it announced System V Release 2 (SVr2) in January 1984. The second revision of the UniForum Draft Standard, in 1984, both tracked and influenced the API of SVr2. Later Unix standards also tended to track System V except in areas where BSD facilities were clearly functionally superior (thus, for example, modern Unix standards describe the System V terminal controls rather than the BSD interface to the same facilities).

In 1985, AT&T released the *System V Interface Definition* (SVID). SVID provided a more formal description of the SVr2 API, incorporating UDS 84. Later revisions SVID2 and SVID3 tracked the interfaces of System V releases 3 and 4. SVID became the basis for the POSIX standards, which ultimately tipped most of the Berkeley/AT&T disputes over system and C library calls in AT&T's favor.

But this would not become obvious for a few years yet; meanwhile, the Unix wars raged on. For example, 1985 saw the release of two competing API standards for file system sharing over networks: Sun's Network File System (NFS) and AT&T's Remote File System (RFS). Sun's NFS prevailed because Sun was willing to share not merely specifications but open-source code with others.

The lesson of this success should have been all the more pointed because on purely logical grounds RFS was the superior model. It supported better file-locking semantics and better mapping among user identities on different systems, and generally made an effort to get the finer details of Unix file system semantics precisely right, unlike NFS. The lesson was ignored, however, even when it was repeated in 1987 by the open-source X windowing system's victory over Sun's proprietary Networked Window System (NeWS).

After 1985 the main thrust of Unix standardization passed to the Institute of Electrical and Electronic Engineers (IEEE). The IEEE's 1003 committee developed a series of standards generally known as POSIX.[144] These went beyond describing merely systems calls and C library facilities; they specified detailed semantics of a shell and a minimum command set, and also detailed bindings for various non-C programming languages. The first release in 1990 was followed by a second edition in 1996. The International Standards Organization adopted them as ISO/IEC 9945.

Key POSIX standards include the following:

1003.1 (released 1990)	Library procedures. Described the C system call API, much like Version 7 except for signals and the terminal-control interface.
1003.2 (released 1992)	Standard shell and utilities. Shell semantics strongly resemble those of the System V Bourne shell.
1003.4 (released 1993)	Real-time Unix. Binary semaphores, process memory locking, memory-mapped files, shared memory, priority scheduling, real-time signals, clocks and timers, IPC message passing, synchronized I/O, asynchronous I/O, real-time files. In the 1996 Second Edition, 1003.4 was split into 1003.1b (real-time) and 1003.1c (threads).

Despite being underspecified in a couple of key areas such as signal-handling semantics and omitting BSD sockets, the original POSIX standards became the basis of all later Unix standardization work. They are still cited as an authority, albeit indirectly through references like *POSIX Programmer's Guide* [Lewine]. The de facto Unix API standard is still "POSIX plus sockets", with later standards mainly adding features and specifying conformance in unusual edge cases more closely.

The next player on the scene was X/Open (later renamed the Open Group), a consortium of Unix vendors formed in 1984. Their X/Open Portability Guides (XPGs) initially developed in parallel with the POSIX drafts, then after 1990 the XPGs incorporated and extended POSIX. Unlike POSIX,

[144] The original 1986 trial-use standard was called IEEE-IX. The name 'POSIX' was suggested by Richard Stallman. The introduction to POSIX.1 says: "It is expected to be pronounced pahz-icks as in positive, not poh-six, or other variations. The pronounciation has been published in an attempt to promulgate a standardized way of referring to a standard operating system interface".

which attempted to capture a safe subset of all Unixes, the XPGs were oriented more toward common practice at the leading edge; even XPG1 in 1985, spanning SVr2 and 4.2BSD, included sockets.

XPG2 in 1987 added a terminal-handling API that was essentially System V curses(3). XPG3 in 1990 merged in the X11 API. XPG4 in 1992 mandated full compliance with the 1989 ANSI C standard. XPG2, 3, and 4 were heavily concerned with support of internationalization and described an elaborate API for handling codesets and message catalogs.

In reading about Unix standards you might come across references to “Spec 1170” (from 1993), “Unix 95” (from 1995) and “Unix 98” (from 1998). These were certification marks based on the X/Open standards; they are now of historical interest only. But the work done on XPG4 turned into Spec 1170, which turned into the first version of the Single Unix Specification (SUS).

In 1993 seventy-five systems and software vendors including every major Unix company put a final end to the Unix wars when they declared backing for X/Open to develop a common definition of Unix. As part of the arrangement, X/Open acquired the rights to the Unix trademark. The merged standard became Single Unix Standard version 1. It was followed in 1997 by a version 2. In 1999 X/Open absorbed the POSIX activity.

In 2001, X/Open (now The Open Group) issued the Single Unix Standard version 3 [http://www.unix.org/version3/]. All the threads of Unix API standardization were finally gathered into one bundle. This reflected facts on the ground; the different varieties of Unix had re-converged on a common API. And, at least among old-timers who remembered the turbulence of the 1980s, there was much rejoicing.

The Ghost at the Victory Banquet

There was, unfortunately, an awkward detail — the old-school Unix vendors who had backed the effort were under severe pressure from the new school of open-source Unixes, and were in some cases in the process of abandoning (in favor of Linux) the proprietary Unixes for which they had gone to so much effort to secure conformance.

The conformance testing needed to verify Single Unix Specification conformance is an expensive proposition. It would need to be done on a per-distribution basis, but is well out of the reach of most distributors of open-source operating systems. In any case, Linux changes so fast that any given release of a distribution would probably be obsolete by the time it could get certified.[145]

[145]One Linux distributor, Lasermoon in Great Britain, did achieve POSIX.1 FIPS 151-2 certification — and went out of business, because potential customers didn’t care.

Standards like the Single Unix Specification have not entirely lost their relevance. They're still valuable guides for Unix implementers. But how The Open Group and other institutions of the old-school Unix standardization process will adapt to the rapid tempo of open-source releases (and to the low- or zero-budget operation of open-source development groups!) remains to be seen.

Unix Standards in the Open-Source World

In the mid-1990s, the open-source community began standardization efforts of its own. These efforts built on the source-code-level compatibility secured by POSIX and its descendants. Linux, in particular, had been written from scratch in a way that depended on the availability of Unix API standards like POSIX.[146]

In 1998 Oracle ported its market-leading database product to Linux, in a move that was rightly seen as a major breakthrough in Linux's mainstream acceptance. The engineer in charge of the port provided a definitive demonstration that API standards had done their job when he was asked by a reporter what technical challenges Oracle had had to surmount. The engineer's reply was “We typed ‘make’.”

The problem for the new-school Unixes, therefore, was not API compatibility at the source-code level. Everybody took for granted the ability to move source code between different Linux, BSD, and proprietary-Unix distributions without more than a trivial amount of porting labor. The new problem was not source compatibility but binary compatibility. For the ground under Unix had shifted in a subtle way as a consequence of the triumph of commodity PC hardware.

In the old days, each Unix had run on what was effectively its own hardware platform. There was enough variety in processor instruction sets and machine architectures that applications had to be ported at source level to move at all. On the other hand, there were a relatively few major Unix releases, each with relatively long service lifetimes. Application vendors like Oracle could afford the cost of building and shipping separate binary distributions for each of three or four hardware/software combinations, because they could amortize the low cost of source-code porting over large customer populations and a long enough product life cycle.

But then the minicomputer and workstation vendors were swamped by inexpensive 386-based supermicros, and open-source Unixes changed the rules. Vendors found they no longer had a stable platform to ship their binaries to.

[146] See *Just for Fun* [Torvalds] for discussion.

The superficial problem, at first, was the large number of Unix distributors — but as the Linux distribution market consolidated, it became clear that the real issue was the rate of change over time. APIs were stable, but the expected locations of system administrative files, utility programs, and things like the prefix of the paths to user mailbox names and system log files kept changing.

The first standards effort to develop within the new-school Linux and BSD community itself (beginning in 1993) was the Filesystem Hierarchy Standard (FHS). This was incorporated into the Linux Standards Base (LSB), which also standardized an expected set of service libraries and helper applications. Both standards became activities of the Free Standards Group [http://www.freestandards.org/], which by 2001 developed a role similar to X/Open's position amidst the old-school Unix vendors.

IETF and the RFC Standards Process

When the Unix community merged with the culture of Internet engineers, it also inherited a mindset formed by the RFC standards process of the Internet Engineering Task Force (IETF). In IETF tradition, standards have to arise from experience with a working prototype implementation — but once they *become* standards, code that does not conform to them is considered broken and mercilessly scrapped.

This is not, sadly, the way standards are normally developed. The history of computing is full of instances in which technical standards were derived by a process that combined the worst features of philosophical axe-grinding with murky back-room politics — producing specifications that failed to resemble anything ever implemented. Worse, many were either so demanding that they could not be practically implemented or so underspecified that they caused more confusion than they resolved. Then they were promulgated to vendors who ignored them wherever they were inconvenient.

One of the more notorious examples of standards nonsense was the Open Systems Interconnect networking protocols that briefly contended with TCP/IP in the 1980s — its 7-layer model looked elegant from a distance but proved overcomplicated and unimplementable in practice.[147] The ANSI X3.64 standard for video-display terminal capabilities is another famous horror story; bedeviled by subtle incompatibilities between legally conformant implementations. Even after character-cell terminals have been largely displaced by bitmapped displays these continue to cause problems (in particular, this is why the function and special keys in your xterm(1) will occasionally break). The RS232 standard for serial communications was so underspecified that it sometimes seemed that no

[147] A Web search is likely to turn up a popular page comparing the OSI 7-layer model with the Taco Bell 7-layer burrito — unfavorably to the former.

two serial cables were alike. Standards horror stories of similar kind could fill a book the size of this one.

The IETF's philosophy has been famously summarized as "We reject kings, presidents, and voting. We believe in rough consensus and running code".[148] That demand for a working implementation first has saved it from the worst category of blunders. In fact its criterion is stronger:

> [A] candidate specification must be implemented and tested for correct operation and interoperability by multiple independent parties and utilized in increasingly demanding environments, before it can be adopted as an Internet Standard.
> — *The Internet Standards Process — Revision 3 (RFC 2026)*

All IETF standards pass through a stage as RFCs (Requests for Comment). The submission process for RFCs is deliberately informal. RFCs may propose standards, survey results, suggest philosophical bases for subsequent RFCs, or even make jokes. The appearance of the annual April 1st RFC is the closest equivalent of a high holy day observance among Internet hackers, and has produced such gems as *A Standard for the Transmission of IP Datagrams on Avian Carriers (RFC 1149)*[149] the *The Hyper Text Coffee Pot Control Protocol (RFC 2324),*[150] and *The Security Flag in the IPv4 Header (RFC 3514)*.[151]

But joke RFCs are about the *only* sort of submission that instantly becomes an RFC. Serious proposals actually start as "Internet-Drafts" floated for public comment via IETF directories on several well-known hosts. Individual Internet-Drafts have no formal status and can be changed or dropped by their originators at any time. If they are neither withdrawn nor promoted to RFC status, they are removed after six months.

Internet-Drafts are not specifications, and software implementers and vendors are specifically barred from claiming compliance with them as if they were specifications. Internet-Drafts are focal points for discussion, usually in a working group connected through an electronic mailing list. When the working group leadership deems fit, the Internet-Draft is submitted to the RFC editor for assignment of an RFC number.

[148] This line was first uttered by senior IETF cadre Dave Clark at the tumultuous 1992 meeting during which the IETF rejected the Open Systems Interconnect protocol.

[149] RFC 1149 is available on the Web [http://www.ietf.org/rfc/rfc1149.txt]. Not only that, it has been implemented [http://www.blug.linux.no/rfc1149/writeup.html].

[150] RFC 2324 is available on the Web [http://www.ietf.org/rfc/rfc2324.txt].

[151] RFC 3514 is available on the Web [http://www.ietf.org/rfc/rfc3514.txt].

Once an Internet-Draft has been published with an RFC number, it is a specification to which implementers may claim conformance. It is expected that the authors of the RFC and the community at large will begin correcting the specification with field experience.

Some RFCs go no further. A specification that fails to attract use and survive field testing can be quietly forgotten, and eventually marked “Not recommended” or “Superseded” by the RFC editor. Failed proposals are accepted as one of the overheads of the process, and no stigma is attached to being associated with one.

The steering committee of the IETF (IESG, or Internet Engineering Steering Group) is responsible for putting successful RFCs on the standards track. They do this by designating the RFC a ‘Proposed Standard’. For the RFC to qualify, the specification must be stable, peer-reviewed, and have attracted significant interest from the Internet community. Implementation experience is not absolutely required before an RFC is given Proposed Standard designation, but it is considered highly desirable, and the IESG may require it if the RFC touches the Internet core protocols or might be otherwise destabilizing.

Proposed Standards are still subject to revision, and may even be withdrawn if the IESG and IETF identify a better solution. They are not recommended for use in “disruption-sensitive environments”—don’t put them in your air-traffic-control systems or on intensive-care equipment.

When there are at least two working, complete, independently originated, and interoperable implementations of a Proposed Standard, the IESG may elevate it to Draft Standard status. RFC 2026 says: “Elevation to Draft Standard is a major advance in status, indicating a strong belief that the specification is mature and will be useful”.

Once an RFC has reached Draft Standard status, it will be changed only to address bugs in the logic of the specification. Draft Standards are expected to be ready for deployment in disruption-sensitive environments.

When a Draft Standard has passed the test of widespread implementation and reached general acceptance, it may be blessed as an Internet Standard. Internet Standards keep their RFC numbers, but also get a number in the STD series. At time of writing there are over 3000 RFCs but only 60 STDs.

RFCs not on standards track may be labeled Experimental, Informational (the joke RFCs get this label), or Historic. The Historic label is applied to obsolete standards. RFC 2026 notes: “(Purists

have suggested that the word should be ‘Historical’; however, at this point, the use of ‘Historic’ is historical.)”

The IETF standards process is designed to encourage standardization driven by practice rather than theory, and to ensure that standard protocols have undergone rigorous peer review and testing. The success of this model is evident in its results — the worldwide Internet.

Specifications as DNA, Code as RNA

Even in the paleolithic period of the PDP-7, Unix programmers had always been more prone than their counterparts elsewhere to treat old code as disposable. This was doubtless a product of the Unix tradition’s emphasis on modularity, which makes it easier to discard and replace small pieces of systems without losing everything. Unix programmers have learned by experience that trying to salvage bad code or a bad design is often more work than rebooting the project. Where in other programming cultures the instinct would be to patch the monster monolith because you have so much work invested in it, the Unix instinct is usually to scrap and rebuild.

The IETF tradition reinforced this by teaching us to think of code as secondary to standards. Standards are what enable programs to cooperate; they knit our technologies into wholes that are more than the sum of the parts. The IETF showed us that careful standardization, aimed at capturing the best of existing practice, is a powerful form of humility that achieves more than grandiose attempts to remake the world around a never-implemented ideal.

After 1980, the impact of that lesson was increasingly widely felt in the Unix community. Thus, while the ANSI/ISO C standard from 1989 is not completely without flaws, it is exceptionally clean and practical for a standard of its size and importance. The Single Unix Specification contains fossils from three decades of experimentation and false starts in a more complicated domain, and is therefore messier than ANSI C. But the component standards it was composed from are quite good; strong evidence for this is the fact that Linus Torvalds successfully built a Unix from scratch by reading them. The IETF’s quiet but powerful example created one of the critical pieces of context that made Linus Torvalds’s feat possible.

Respect for published standards and the IETF process has become deeply ingrained in the Unix culture; deliberately violating Internet STDs is simply Not Done. This can sometimes create chasms of mutual incomprehension between people with a Unix background and others prone to assume that the most popular or widely deployed implementation of a protocol is by definition correct — even if it breaks the standard so severely that it will not interoperate with properly conforming software.

The Unix programmer's respect for published standards is more interesting because he is likely to be rather hostile to a-priori specifications of other kinds. By the time the 'waterfall model' (specify exhaustively first, then implement, then debug, with no reverse motion at any stage) fell out of favor in the software-engineering literature, it had been an object of derision among Unix programmers for years. Experience, and a strong tradition of collaborative development, had already taught them that prototyping and repeated cycles of test and evolution are a better way.

The Unix tradition clearly recognizes that there can be great value in good specifications, but it demands that they be treated as provisional and subject to revision through field experience in the way that Internet-Drafts and Proposed Standards are. In best Unix practice, the documentation of the program is used as a specification subject to revision analogously to an Internet Proposed Standard.

> Unlike other environments, in Unix development the documentation is often written before the program, or at least in conjunction with it. For X11, the core X standards were finished before the first release of X and have remained essentially unchanged since that date. Compatibility among different X systems is improved further by rigorous specification-driven testing.
>
> The existence of a well-written specification made the development of the X test suite much easier. Each statement in the X specification was translated into code to test the implementation, a few minor inconsistencies were uncovered in the specification during this process, but the result is a test suite that covers a significant fraction of the code paths within the sample X library and server, and all without referring to the source code of that implementation.
>
> —
>
> <author>KeithPackard</author>

Semiautomation of the test-suite generation proved to be a major advantage. While field experience and advances in the state of the graphics art led many to criticize X on design grounds, and various portions of it (such as the security and user-resource models) came to seem clumsy and over-engineered, the X implementation achieved a remarkable level of stability and cross-vendor interoperation.

In Chapter 9 we discussed the value of pushing coding up to the highest possible level to minimize the effects of constant defect density. Implicit in Keith Packard's account is the idea that the X documentation constituted no mere wish-list but a form of high-level code. Another key X developer confirms this:

> In X, the specification has always ruled. Sometimes specs have bugs that need to be fixed too, but code is usually buggier than specs (for any spec worth its ink, anyway).
>
> —
>
> <author>JimGettys</author>

Jim goes on to observe that X's process is actually quite similar to the IETF's. Nor is its utility limited to constructing good test suites; it means that arguments about the system's behavior can be conducted at a functional level with respect to the specification, avoiding too much entanglement in implementation issues.

> Having a well-considered specification driving development allows for little argument about bug vs. feature; a system which incorrectly implements the specification is broken and should be fixed.
>
> I suspect this is so ingrained into most of us that we lose sight of its power.
>
> A friend of mine who worked for a small software firm east of Bellevue wondered how Linux applications developers could get OS changes synchronized with application releases. In that company, major system-level APIs change frequently to accommodate application whims and so essential OS functionality must often be released along with each application.
>
> I described the power held by the specifications and how the implementation was subservient to them, and then went on to assert that an application which got an unexpected result from a documented interface was either broken or had discovered a bug. He found this concept startling.
>
> Discerning such bugs is a simple matter of verifying the implementation of the interface against the specification. Of course, having source for the implementation makes that a bit easier.
>
> —
>
> <author>KeithPackard</author>

This standards-come-first attitude has benefits for end users as well. While that no-longer-small company east of Bellevue has trouble keeping its office suite compatible with its own previous releases, GUI applications written for X11 in 1988 still run without change on today's X

implementations. In the Unix world, this sort of longevity is normal — and the standards-as-DNA attitude is the reason why.

Thus, experience shows that the standards-respecting, scrap-and-rebuild culture of Unix tends to yield better interoperability over extended time than perpetual patching of a code base without a standard to provide guidance and continuity. This may, indeed, be one of the most important Unix lessons.

Keith's last comment brings us directly to an issue that the success of open-source Unixes has brought to the forefront — the relationship between open standards and open source. We'll address this at the end of the chapter — but before doing that, it's time to address the practical question of how Unix programmers can actually *use* the tremendous body of accumulated standards and lore to achieve software portability.

Programming for Portability

Software portability is usually thought of in quasi-spatial terms: can this code be moved sideways to existing hardware and software platforms other than the one it was built for? But Unix experience over decades tells us that durability down through time is just as important, if not more so. If we could predict the future of software in detail it would probably be the present — nevertheless, in programming for portability we should try to think about making choices that will base the software on the features of its environment that are likeliest to persist, and avoid technologies that seem likely to face end-of-life in the foreseeable future.

Under Unix, two decades of attention to the issues of specifying portable APIs has largely solved that problem. Facilities described in the Single Unix Specification are likely to be present on all modern Unix platforms today and rather unlikely to go unsupported in the future.

But not all platform dependencies have to do with the system or library APIs. Your implementation language can matter; file-system layout and configuration differences between the source and target system can be a problem as well. But Unix practice has evolved ways to cope.

Portability and Choice of Language

The first issue in programming for portability is your choice of implementation language. All the major languages we surveyed in Chapter 14 are highly portable in the sense that compatible implementations are available across all modern Unixes; for most, implementations under Windows and MacOS are available as well. Portability problems tend to arise not in the core languages

but in support libraries and degree of integration with the local environment (especially IPC and concurrent-process management, including the infrastructure for GUIs).

C Portability

The core C language is extremely portable. The standard Unix implementation is the GNU C compiler, which is ubiquitous not only in open-source Unixes but modern proprietary Unixes as well. GNU C has been ported to Windows and classic MacOS, but is not widely used in either environment because it lacks portable bindings to the native GUI.

The standard I/O library, mathematics routines, and internationalization support are portable across all C implementations. File I/O, signals, and process control are portable across Unixes provided one takes care to use only the modern APIs described in the Single Unix Specification; older C code often has thickets of preprocessor conditionals for portability, but those handle legacy pre-POSIX interfaces from older proprietary Unixes that are obsolete or close to it in 2003.

C portability starts to be a more serious problem near IPC, threads, and GUI interfaces. We discussed IPC and threads portability issues in Chapter 7. The real practical problem is GUI toolkits. A number of open-source GUI toolkits are universally portable across modern Unixes and to Windows and classic MacOS as well — Tk, wxWindows, GTK, and Qt are four well-known ones with source code and documentation readily discoverable by Web search. But none of them is shipped with all platforms, and (for reasons more legal than technical) none of these offers the native-GUI look and feel on all platforms. We gave some guidelines for coping in Chapter 15.

Volumes have been written on the subject of how to write portable C code. This book is not going to be one of them. Instead, we recommend a careful reading of *Recommended C Style and Coding Standards* [Cannon] and the chapter on portability in *The Practice of Programming* [Kernighan-Pike99].

C++ Portability

C++ has all the same operating-system-level portability issues as C, and some of its own. An additional one is that the open-source GNU compiler for C++ has lagged substantially behind the proprietary implementations for most of its existence; thus, there is not yet as of mid-2003 a universally deployed equivalent of GNU C on which to base a de-facto standard. Furthermore, no C++ compiler yet implements the full C++99 ISO standard for the language, though GNU C++ comes closest.

Shell Portability

Shell-script portability is, unfortunately, poor. The problem is not shell itself; bash(1) (the open-source Bourne Again shell) has become sufficiently ubiquitous that pure shellscripts can run almost anywhere. The problem is that most shellscripts make heavy use of other commands and filters that are much less portable, and by no means guaranteed to be in the toolkit in any given target machine.

This problem can be overcome by dint of heroic effort, as in the autoconf(1) tools. But it is sufficiently severe that most of the heavier sort of programming that used to be done in shell has moved to second-generation scripting languages like Perl, Python, and Tcl.

Perl Portability

Perl has good portability. Stock Perl even offers a portable set of bindings to the Tk toolkit that supports portable GUIs across Unix, MacOS and Windows. One issue dogs it, however. Perl scripts often require add-on libraries from CPAN (the Comprehensive Perl Archive Network) which are not guaranteed to be present with every Perl implementation.

Python Portability

Python has excellent portability. Like Perl, stock Python even offers a portable set of bindings to the Tk toolkit that supports portable GUIs across Unix, MacOS, and Windows.

Stock Python has a much richer standard library than does Perl and no equivalent of CPAN for programmers to rely on; instead, important extension modules are routinely incorporated into the stock Python distribution during minor releases. This trades a spatial problem for a temporal one, making Python much less subject to the missing-module effect at the cost of making Python minor version numbers somewhat more important than Perl release levels are. In practice, the tradeoff seems to favor Python.

Tcl Portability

Tcl portability is good, overall, but varies sharply by project complexity. The Tk toolkit for cross-platform GUI programming is native to Tcl. As with Python, evolution of the core language has been relatively smooth, with few version-skew problems. Unfortunately, Tcl relies even more heavily than Perl on extension facilities that are not guaranteed to ship with every implementation — and there is no equivalent of CPAN to centrally distribute them.

For smaller projects not reliant on extensions, therefore, Tcl portability is excellent. But larger projects tend to depend heavily on both extensions and (as with shell programming) calling external commands that may or may not be present on the target machine; their portability tends to be poor.

Tcl may have suffered, ironically, from the ease of adding extensions to it. By the time a particular extension started to look interesting as part of the standard distribution, there typically were several different versions of it in existence. At the 1995 Tcl/Tk Workshop, John Osterhout explained why there was no OO support in the standard Tcl distribution:

> Think of five mullahs sitting around in a circle, all saying “Kill him, he’s a heathen”. If I put a specific OO scheme into the core, then one of them will say “Bless you, my son, you may kiss my ring”, and the other four will say “Kill him, he’s a heathen”.

The lot of a language designer is not necessarily a happy one.

Java Portability

Java portability is excellent — it was, after all, designed with “write once, run everywhere” as a primary goal. Portability fails, however, to be perfect. The difficulties are mostly version-skew problems between JDK 1.1 and the older AWT GUI toolkit (on the one hand) and JDK 1.2 with the newer Swing GUI toolkit. There are several important reasons for these:

- Sun’s AWT design was so deficient that it had to be replaced with Swing.
- Microsoft’s refusal to support Java development on Windows and attempt to replace it with C#.
- Microsoft’s decision to hold Internet Explorer’s applet support at the JDK 1.1 level.
- Sun licensing terms that make open-source implementations of JDK 1.2 impossible, retarding its deployment (especially in the Linux world).

For programs that involve GUIs, Java developers seeking portability will, for the foreseeable future, face a choice: Stay in JDK 1.1/AWT with a poorly designed toolkit for maximum portability (including to Microsoft Windows), or get the better toolkit and capabilities of JDK 1.2 at the cost of sacrificing some portability.

Finally, as we noted previously, the Java thread support has portability problems. The Java API, unlike less ambitious operating-system bindings for other languages, bravely tried to bridge the gaps between the diverging process models offered by different operating systems. It does not quite manage the trick.

Emacs Lisp Portability

Emacs Lisp portability is excellent. Emacs installations tend to be upgraded frequently, so seriously out-of-date environments are rare. The same extension Lisp is supported everywhere and effectively all extensions are shipped with Emacs itself.

Then, too, the primitive set of Emacs is quite stable. It achieved completeness for the things an editor has to do (manipulating buffers, bashing text) years ago. Only the introduction of X has disturbed this picture at all, and very few Emacs modes need to be aware of X. Portability problems are usually manifestations of quirks in the C-level bindings of operating-system facilities; control of subordinate processes in modes like mail agents is about the only issue where such problems manifest with any frequency.

Avoiding System Dependencies

Once your language and support libraries are chosen, the next portability issue is usually the location of key system files and directories: mail spools, logfile directories and the like. The archetype of this sort of problem is whether the mail spool directory is `/var/spool/mail` or `/var/mail`.

Often, you can avoid this sort of dependency by stepping back and reframing the problem. Why are you opening a file in the mail spool directory, anyway? If you're writing to it, wouldn't it be better to simply invoke the local mail transport agent to do it for you so the file-locking gets done right? If you're reading from it, might it be better to query it through a POP3 or IMAP server?

The same sort of question applies elsewhere. If you find yourself opening logfiles manually, shouldn't you be using syslog(3) instead? Function-call interfaces through the C library are better standardized than system file locations. Use that fact!

If you must have system file locations in your code, your best alternative depends on whether you will be distributing in source code or binary form. If you are distributing in source, the *autoconf* tools we discuss in the next section will help you. If you're distributing in binary, then it's good practice to have your program poke around at runtime and see if it can automatically adapt itself to local conditions — say, by actually checking for the existence of `/var/mail` and `/var/spool/mail`.

Tools for Portability

You can often use the open-source GNU autoconf(1) we surveyed in Chapter 15 to handle portability issues, do system-configuration probes, and tailor your makefiles. People building from sources today expect to be able to type **configure; make; make install** and get a clean build. There is a good tutorial on these tools [http://seul.org/docs/autotut/]. Even if you're distributing in binary, the autoconf(1) tools can help automate away the problem of conditionalizing your code for different platforms.

Other tools that address this problem; two of the better known are the Imake(1) tool associated with the X windowing system and the *Configure* tool built by Larry Wall (later the inventor of Perl) and adapted for many different projects. All are at least as complicated as the autoconf suite, and no longer as often used. They don't cover as wide a range of target systems.

Internationalization

An in-depth discussion of code internationalization — designing software so the interface readily incorporates multiple languages and the vagaries of different character sets — would be out of scope for this book. However, a few lessons for good practice do stand out from Unix experience.

First, *separate the message base from the code*. Good Unix practice is to separate the message strings a program uses from its code. so that message dictionaries in other languages can be plugged in without modifying the code.

The best-known tool for this job is GNU *gettext*, which requires that you wrap native-language strings that need to be internationalized in a special macro. The macro uses each string as a key into per-language dictionaries which can be supplied as separate files. If no such dictionaries are available (or if they are but the string lookup does not return a match), the macro simply returns its argument, implicitly falling back on the native language in the code.

While *gettext* itself is messy and fragile as of mid-2003, its general philosophy is sound. For many projects, it is possible to craft a lighter-weight version of this idea with good results.

Second, there is a clear trend in modern Unixes to scrap all the historical cruft associated with multiple character sets and make applications natively speak UTF-8, the 8-bit shift encoding of the Unicode character set (as opposed to, say, making them natively speak 16-bit wide characters). The low 128 characters of UTF-8 are ASCII, and the low 256 are Latin-1, which means this choice is backward-compatible with the two most widely used character sets. The fact that XML and Java have made this choice helps, but the momentum is present even where XML and Java are not.

Third, beware of character ranges in regular expressions. The element **[a-z]** will not necessarily catch all lower-case letters if the script or program it's in is applied to (say) German, where the sharp-s or ß character is considered lower-case but does not fall in that range; similar problems arise with French accented letters. Its safer to use **[[:lower:]]**. and other symbolic ranges described in the POSIX standard.

Portability, Open Standards, and Open Source

Portability requires standards. Open-source reference implementations are the most effective method known for both promulgating a standard and for pressuring proprietary vendors into conforming. If you are a developer, open-source implementations of a published standard can both tremendously reduce your coding workload and allow your product to benefit (in ways both expected and unexpected) from the labor of others.

Let's suppose, for example, you are designing image-capture software for a digital camera. Why write your own format for saving image bits or buy proprietary code when (as we noted in Chapter 5) there is a well-tested, full-featured library for writing PNGs in open source?

The (re)invention of open source has had a significant impact on the standards process as well. Though it is not formally a requirement, the IETF has since around 1997 grown increasingly resistant to standard-tracking RFCs that do not have at least one open-source reference implementation. In the future, it seems likely that conformance to any given standard will increasingly be measured by conformance to (or outright use of!) open-source implementations that have been blessed by the standard's authors.

> The flip side of this is that often the best way to *make* something a standard is to distribute a high-quality open-source implementation of it.
>
> —

<author>HenrySpencer</author>

In the end, the most effective step you can take to ensure the portability of your code is to not rely on proprietary technology. You never know when the closed-source library or tool or code generator or network protocol you are depending on will be end-of-lifed, or when the interface will be changed in some backwards-incompatible way that breaks your project. With open-source code, you have a path forward even if the leading-edge version changes in a way that breaks your project; because you have access to source code, you can forward-port it to new platforms if you need to.

Until the late 1990s this advice would have been impractical. The few alternatives to relying on proprietary operating systems and development tools were noble experiments, academic proofs-of-concept, or toys. But the Internet changed everything; in mid-2003 Linux and the other open-source Unixes exist and have proven their mettle as platforms for delivering production-quality software. Developers have a better option now than being dependent on short-term business decisions designed to protect someone else's monopoly. Practice defensive design — build on open source and don't get stranded!

Chapter 18. Documentation

Explaining Your Code to a Web-Centric World

I've never met a human being who would want to read 17,000 pages of documentation, and if there was, I'd kill him to get him out of the gene pool.
--
<author>JosephCostello</author>

Unix's first application, in 1971, was as a platform for document preparation — Bell Labs used it to prepare patent documents for filing. Computer-driven phototypesetting was still a novel idea then, and for years after it debuted in 1973 Joe Ossana's troff(1) formatter defined the state of the art.

Ever since, sophisticated document formatters, typesetting software, and page-layout programs of various sorts have been an important theme in the Unix tradition. While troff(1) has proven surprisingly durable, Unix has also hosted a lot of groundbreaking work in this application area. Today, Unix developers and Unix tools are at the cutting edge of far-reaching changes in documentation practice triggered by the advent of the World Wide Web.

At the user-presentation level, Unix-community practice has been moving rapidly toward 'everything is HTML, all references are URLs' since the mid-1990s. Increasingly, modern Unix help browsers are simply Web browsers that know how to parse certain specialized kinds of URLs (for example, 'man:ls(1)' interprets the ls(1) man page into HTML). This relieves the problems created by having lots of different formats for documentation masters, but does not entirely solve them. Documentation composers still have to grapple with issues about which master format best meets their particular needs.

In this chapter, we'll survey the rather unfortunate surfeit of different documentation formats and tools left behind by decades of experimentation, and we'll develop guidelines for good practice and good style.

Documentation Concepts

Our first distinction is between "What You See Is What You Get" (WYSIWYG) documentation programs and *markup-centered tools*. Most desktop-publishing programs and word processors are in the former category; they have GUIs in which what one types is inserted directly into an on-screen presentation of the document intended to resemble the final printed version as closely as possible. In

a markup-centered system, by contrast, the master document is normally flat text containing explicit, visible control tags and not at all resembling the intended output. The marked-up source can be modified with an ordinary text editor, but has to be fed to a formatter program to produce rendered output for printing or display.

The visual-interface, WYSIWYG style was too expensive for early computer hardware, and remained rare until the advent of the Macintosh personal computer in 1984. It is completely dominant on non-Unix operating systems today, Native Unix document tools, on the other hand, are almost all markup-centered. The Unix troff(1) of 1971 was a markup formatter, and is probably the oldest such program still in use.

Markup-centered tools still have a role because actual implementations of WYSIWYG tend to be broken in various ways — some superficial, some deep. WYSIWYG document processors have the general problem with GUIs that we discussed in Chapter 11; the fact that you *can* visually manipulate anything tends to mean you *must* visually manipulate everything. That would remain a problem even if the WYSIWIG correspondence between screen and printer output were perfect — but it almost never is.

In truth, WYSIWYG document processors aren't exactly WYSIWIG. Most have interfaces that obscure the differences between screen presentation and printer output without actually eliminating them. Thus they violate the Rule of Least Surprise: the visual aspect of the interface encourages you to use the program like a typewriter even though it is not, and your input will occasionally produce an unexpected and undesired result.

In further truth, WYSIWIG systems actually rely on markup codes but expend great effort on keeping them invisible in normal use. Thus they break the Rule of Transparency: you can't see all of the markup, so it is difficult to fix documents that break because of misplaced markup codes.

Despite its problems, WYSIWYG document processors can be very nice if what you want is to slide a picture three ems to the right on the cover of a four-page brochure. But they tend to be constricting any time you need to make a global change to the layout of a 300-page manuscript. WYSIWYG users faced with that kind of challenge must give it up or suffer the death of a thousand mouse clicks; in situations like that, there is really no substitute for being able to edit explicit markup, and Unix's markup-centered document tools offer better solutions.

Today, in a world influenced by the example of the Web and XML, it has become common to make a distinction between *presentation* and *structural* markup in documents — the former being instructions about how a document should look, the latter being instructions about how it's organized

and what it means. This distinction wasn't clearly understood or followed through in early Unix tools, but it's important for understanding the design pressures that led to today's descendants of them.

Presentation-level markup carries all the formatting information (e.g., about desired whitespace layout and font changes) in the document itself. In a structural-markup system, the document has to be combined with a *stylesheet* that tells the formatter how to translate the structure markup in the document to a physical layout. Both kinds of markup ultimately control the physical appearance of a printed or browsed document, but structural markup does it through one more level of indirection that turns out to be necessary if you want to produce good results for both printing and the Web.

Most markup-centered documentation systems support a macro facility. Macros are user-defined commands that are expanded by text substitution into sequences of built-in markup requests. Usually, these macros add structural features (like the ability to declare section headings) to the markup language.

> The troff macro sets (*mm*, *me*, and my *ms* package) were actually designed to push people away from format-oriented editing and toward content-oriented editing. The idea was to label the semantic parts and then have different style packages that would know whether in this style the title should be boldfaced or not, centered or not, and so on. Thus there was at one point a set of macros that tried to imitate ACM style, and another that imitated Physical Review style, but used the basic `-ms` markup. All of the macros lost out to people who were focused on producing one document, and controlling its appearance, just as Web pages get bogged down in the dispute over whether the reader or author should control the appearance. I frequently found secretaries who were using the .AU (author name) command just to produce italics, noticing that it did that, and then getting into trouble with its other effects.
>
> —
>
> <author>MikeLesk</author>

Finally, we note that there are significant differences between the sorts of things composers want to do with small documents (business and personal letters, brochures, newsletters) and the things they want to do with large ones (books, long articles, technical papers, and manuals). Large documents tend to have more structure, to be pieced together from parts that may have to be changed separately, and to need automatically-generated features like tables of contents; these are both traits that favor markup-centered tools.

The Unix Style

The Unix style of documentation (and documentation tools) has several technical and cultural traits that set it apart from the way documentation is done elsewhere. Understanding these signature traits first will create context for you to understand why the programs and the practice look the way they do, and why the documentation reads the way it does.

The Large-Document Bias

Unix documentation tools have always been designed primarily for the challenges involved in composing large and complex documents. Originally it was patent applications and paperwork; later it was scientific and technical papers, technical documentation of all sorts. Consequently, most Unix developers learned to love markup-centered documentation tools. Unlike the PC users of the time, the Unix culture was unimpressed with WYSIWYG word processors when they became generally available in the late 1980s and early 1990s — and even among today's younger Unix hackers it is still unusual to find anyone who really prefers them.

Dislike of opaque binary document formats — and especially of opaque *proprietary* binary formats — also played a part in the rejection of WYSIWYG tools. On the other hand, Unix programmers seized on PostScript (the now-standard language for controlling imaging printers) with enthusiasm as soon as the language documentation became available; it fits neatly in the Unix tradition of domain-specific languages. Modern open-source Unix systems have excellent PostScript and Portable Document Format (PDF) tools.

Another consequence of this history is that Unix documentation tools have tended to have relatively weak support for including images, but strong support for diagrams, tables, graphing, and mathematical typesetting — the sorts of things often needed in technical papers.

The Unix attachment to markup-centered systems has often been caricatured as a prejudice or a troglodyte trait, but it is not really anything of the kind. Just as the putatively 'primitive' CLI style of Unix is in many ways better adapted to the needs of power users than GUIs, the markup-centered design of tools like troff(1) is a better fit for the needs of power documenters than are WYSIWYG programs.

The large-document bias in Unix tradition did not just keep Unix developers attached to markup-based formatters like *troff*, it also made them interested in structural markup. The history of Unix document tools is one of lurching, muddled, and erratic movement in a general direction away from

presentation markup and toward structural markup. In mid-2003 this journey is not yet over, but the end is distantly in sight.

The development of the World Wide Web meant that the ability to render documents in multiple media (or, at least, for both print and HTML display) became the central challenge for documentation tools after about 1993. At the same time, even ordinary users were, under the influence of HTML, becoming more comfortable with markup-centered systems. This led directly to an explosion of interest in structural markup and the invention of XML after 1996. Suddenly the old-time Unix attachment to markup-centered systems started looking prescient rather than reactionary.

Today, in mid-2003, most of the leading-edge development of XML-based documentation tools using structural markup is taking place under Unix. But, at the same time, the Unix culture has yet to let go of its older tradition of presentation-level markup systems. The creaking, clanking, armor-plated dinosaur that is *troff* has only partly been displaced by HTML and XML.

Cultural Style

Most software documentation is written by technical writers for the least-common-denominator ignorant — the knowledgeable writing for the knowledgeless. The documentation that ships with Unix systems has traditionally been written by programmers for their peers. Even when it is not peer-to-peer documentation, it tends to be influenced in style and format by the enormous mass of programmer-to-programmer documentation that ships with Unix systems.

The difference this makes can be summed up in one observation: Unix manual pages traditionally have a section called BUGS. In other cultures, technical writers try to make the product look good by omitting and skating over known bugs. In the Unix culture, peers describe the known shortcomings of their software to each other in unsparing detail, and users consider a short but informative BUGS section to be an encouraging sign of quality work. Commercial Unix distributions that have broken this convention, either by suppressing the BUGS section or euphemizing it to a softer tag like LIMITATIONS or ISSUES or APPLICATION USAGE, have invariably fallen into decline.

Where most other software documentation tends to to oscillate between incomprehensibility and oversimplifying condescension, classic Unix documentation is written to be telegraphic but complete. It does not hold you by the hand, but it usually points in the right direction. The style assumes an active reader, one who is able to deduce obvious unsaid consequences of what is said, and who has the self-confidence to trust those deductions.

Unix programmers tend to be good at writing references, and most Unix documentation has the flavor of a reference or *aide memoire* for someone who thinks like the document-writer but is not yet an expert at his or her software. The results often look much more cryptic and sparse than they actually are. Read every word carefully, because whatever you want to know will probably be there, or deducible from what's there. Read every word carefully, because you will seldom be told anything twice.

The Zoo of Unix Documentation Formats

All the major Unix documentation formats except the very newest one are presentation-level markups assisted by macro packages. We examine them here from oldest to newest.

troff and the Documenter's Workbench Tools

We discussed the Documenter's Workbench architecture and tools in Chapter 8 as an example of how to integrate a system of multiple minilanguages. Now we return to these tools in their functional role as a typesetting system.

The *troff* formatter interprets a presentation-level markup language. Recent implementations like the GNU project's groff(1) emit PostScript by default, though it is possible to get other forms of output by selecting a suitable driver. See Example 18.1 for several of the *troff* codes you might encounter in document sources.

Example 18.1. groff(1) markup example.

```
This is running text.
.\" Comments begin with a backslash and double quote.
.ft B
This text will be in bold font.
.ft R
This text will be back in the default (Roman) font.
These lines, going back to "This is running text", will
be formatted as a filled paragraph.
.bp
The bp request forces a new page and a paragraph break.
This line will be part of the second filled paragraph.
.sp 3
The .sp request emits the number of blank lines given as argument
```

```
.nf
The nf request switches off paragraph filling.
Until the fi request switches it back on
whitespace and layout will be preserved.

One word in this line will be in \fBbold\fR font.
.fi

Paragraph filling is back on.
```

troff(1) has many other requests, but you are unlikely to see most of them directly. Very few documents are written in bare *troff*. It supports a macro facility, and half a dozen macro packages are in more or less general use. Of these, the overwhelmingly most common is the man(7) macro package used to write Unix manual pages. See Example 18.2 for a sample.

Example 18.2. *man* markup example.

```
.SH SAMPLE SECTION
The SH macro starts a section, boldfacing the section title.
.P
The P request starts a new paragraph.  The I request sets its
argument in
.I italics.
.IP *
This starts an indented paragraph with an asterisk label.
More text for the first bulleted paragraph.
.TP
This first line will become a paragraph label
This will be the first line in the paragraph, further indented
relative to the label.

The blank line just above this is treated almost exactly like a
paragraph break (actually, like the troff-level request .sp 1).
.SS A subsection
This is subsection text.
```

Two of the other half-dozen historical *troff* macro libraries, ms(7) and mm(7) are still in use. BSD Unix has its own elaborate extended macro set, mdoc(7). All these are designed for writing technical

manuals and long-form documentation. They are similar in style but more elaborate than man macros, and oriented toward producing typeset output.

A minor variant of troff(1) called nroff(1) produces output for devices that can only support constant-width fonts, like line printers and character-cell terminals. When you view a Unix manual page within a terminal window, it is nroff that has rendered it for you.

The Documenter's Workbench tools do the technical-documentation jobs they were designed for quite well, which is why they have remained in continuous use for more than thirty years while computers increased a thousandfold in capacity. They produce typeset text of reasonable quality on imaging printers, and can throw a tolerable approximation of a formatted manual page on your screen.

They fall down badly in a couple of areas, however. Their stock selection of available fonts is limited. They don't handle images well. It's hard to get precise control of the positioning of text or images or diagrams within a page. Support for multilingual documents is nonexistent. There are numerous other problems, some chronic but minor and some absolute showstoppers for specific uses. But the most serious problem is that because so much of the markup is presentation level, it's difficult to make good Web pages out of unmodified *troff* sources.

Nevertheless, at time of writing man pages remain the single most important form of Unix documentation.

TeX

TeX (pronounced */teH/* with a rough h as though you are gargling) is a very capable typesetting program that, like the Emacs editor, originated outside the Unix culture but is now naturalized in it. It was created by noted computer scientist Donald Knuth when he became impatient with the quality of typography, and especially mathematical typesetting, that was available to him in the late 1970s.

TeX, like troff(1), is a markup-centered system. *TeX*'s request language is rather more powerful than *troff*'s; among other things, it is better at handling images, page-positioning content precisely, and internationalization. *TeX* is particularly good at mathematical typesetting, and unsurpassed at basic typesetting tasks like kerning, line filling, and hyphenating. *TeX* has become the standard submission format for most mathematical journals. It is actually now maintained as open source by a working group of the the American Mathematical Society. It is also commonly used for scientific papers.

As with troff(1), human beings usually do not write large volumes of raw *TeX* macros by hand; they use macro packages and various auxiliary programs instead. One particular macro package, *LaTeX*, is almost universal, and most people who say they're composing in *TeX* almost always actually mean they're writing *LaTeX*. Like *troff*'s macro packages, a lot of its requests are semi-structural.

One important use of *TeX* that is normally hidden from the user is that other document-processing tools often generate *LaTeX* to be turned into PostScript, rather than attempting the much more difficult job of generating PostScript themselves. The xmlto(1) front end that we discussed as a shell-programming case study in Chapter 14 uses this tactic; so does the XML-DocBook toolchain we'll examine later in this chapter.

TeX has a wider application range than troff(1) and is in most ways a better design. It has the same fundamental problems as *troff* in an increasingly Web-centric world; its markup has strong ties to the presentation level, and automatically generating good Web pages from *TeX* sources is difficult and fault-prone.

TeX is never used for Unix system documentation and only rarely used for application documentation; for those purposes, *troff* is sufficient. But some software packages that originated in academia outside the Unix community have imported the use of *TeX* as a documentation master format; the Python language is one example. As we noted above, it is also heavily used for mathematical and scientific papers, and will probably dominate that niche for some years yet.

Texinfo

Texinfo is a documentation markup invented by the Free Software Foundation and used mainly for GNU project documentation — including the documentation for such essential tools as Emacs and the GNU Compiler Collection.

Texinfo was the first markup system specifically designed to support both typeset output on paper and hypertext output for browsing. The hypertext format was not, however, HTML; it was a more primitive variety called 'info', originally designed to be browsed from within Emacs. On the print side, Texinfo turns into *TeX* macros and can go from there to PostScript.

The Texinfo tools can now generate HTML. But they don't do a very good or complete job, and because a lot of Texinfo's markup is at presentation level it is doubtful that they ever will. As of mid-2003, the Free Software Foundation is working on heuristic Texinfo to DocBook translation. Texinfo will probably remain a live format for some time.

POD

Plain Old Documentation is the markup system used by the maintainers of Perl. It generates manual pages, and has all the familiar problems of presentation-level markups, including trouble generating good HTML.

HTML

Since the World Wide Web entered the mainstream in the early 1990s, a small but increasing percentage of Unix projects have been writing their documentation directly in HTML. The problem with this approach is that it is difficult to generate high-quality typeset output from HTML. There are particular problems with indexing as well; the information needed to generate indexes is not present in HTML.

DocBook

DocBook is an SGML and XML document type definition designed for large, complex technical documents. It is alone among the markup formats used in the Unix community in being purely structural. The xmlto(1) tool discussed in Chapter 14 supports rendering to HTML, XHTML, PostScript, PDF, Windows Help markup, and several less important formats.

Several major open-source projects (including the Linux Documentation Project, FreeBSD, Apache, Samba, GNOME, and KDE) already use DocBook as a master format. This book was written in XML-DocBook.

DocBook is a large topic. We'll return to it after summing up the problems with the current state of Unix documentation.

The Present Chaos and a Possible Way Out

Unix documentation is, at present, a mess.

Between *man*, *ms*, *mm*, *TeX*, Texinfo, POD, HTML, and DocBook, the documentation master files on modern Unix systems are scattered across eight different markup formats. There is no uniform way to view all the rendered versions. They aren't Web-accessible, and they aren't cross-indexed.

Many people in the Unix community are aware that this is a problem. At time of writing most of the effort toward solving it has come from open-source developers, who are more actively interested

in competing for acceptance by nontechnical end users than developers for proprietary Unixes have been. Since 2000, practice has been moving toward use of XML-DocBook as a documentation interchange format.

The goal, which is within sight but will take a lot of effort to achieve, is to equip every Unix system with software that will act as a systemwide document registry. When system administrators install packages, one step will be to enter the package's XML-DocBook documentation into the registry. It will then be rendered into a common HTML document tree and cross-linked to the documentation already present.

Early versions of the document-registry software are already working. The problem of forward-converting documentation from the other formats into XML-DocBook is a large and messy one, but the conversion tools are falling into place. Other political and technical problems remain to be attacked, but are probably solvable. While there is not as of mid-2003 a communitywide consensus that the older formats have to be phased out, that seems the likeliest working out of events.

Accordingly, we'll next take a very detailed look at DocBook and its toolchain. This description should be read as an introduction to XML under Unix, a pragmatic guide to practice and as a major case study. It's a good example of how, in the context of the Unix community, cooperation between different project groups develops around shared standards.

DocBook

A great many major open-source projects are converging on DocBook as a standard format for their documentation. The advocates of XML-based markup seem to have won the theoretical argument against presentation-level and for structural-level markup, and an effective XML-DocBook toolchain is available in open source.

Nevertheless, a lot of confusion still surrounds DocBook and the programs that support it. Its devotees speak an argot that is dense and forbidding even by computer-science standards, slinging around acronyms that have no obvious relationship to the things you need to do to write markup and make HTML or PostScript from it. XML standards and technical papers are notoriously obscure. In the rest of this section, we'll try to dispel the fog of jargon.

Document Type Definitions

(Note: to keep the explanation simple, most of this section tells some lies, mainly by omitting a lot of history. Truthfulness will be fully restored in a following section.)

DocBook is a structural-level markup language. Specifically, it is a dialect of XML. A DocBook document is a piece of XML that uses XML tags for structural markup.

For a document formatter to apply a stylesheet to your document and make it look good, it needs to know things about the overall structure of your document. For example, in order to physically format chapter headers properly, it needs to know that a book manuscript normally consists of front matter, a sequence of chapters, and back matter. In order for it to know this sort of thing, you need to give it a *Document Type Definition* or DTD. The DTD tells your formatter what sorts of elements can be in the document structure, and in what order they can appear.

What we mean by calling DocBook a 'dialect' of XML is actually that DocBook is a DTD — a rather large DTD, with somewhere around 400 tags in it.[152]

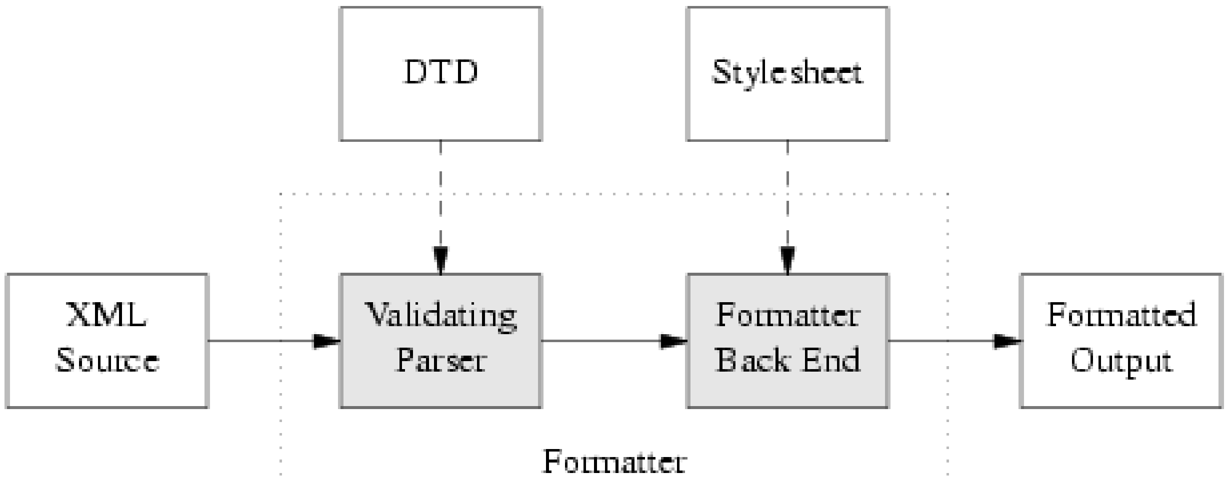
Lurking behind DocBook is a kind of program called a *validating parser*. When you format a DocBook document, the first step is to pass it through a validating parser (the front end of the DocBook formatter). This program checks your document against the DocBook DTD to make sure you aren't breaking any of the DTD's structural rules (otherwise the back end of the formatter, the part that applies your stylesheet, might become quite confused).

The validating parser will either throw an error, giving you messages about places where the document structure is broken, or translate the document into a stream of XML elements and text that the parser back end combines with the information in your stylesheet to produce formatted output.

Figure 18.1 diagrams the whole process.

[152]In XML-speak, what we have been calling a 'dialect' is called an 'application'; we've avoided that usage, since it collides with another more common sense of the word.

Figure 18.1. Processing structural documents.



The part of the diagram inside the dotted box is your formatting software, or *toolchain*. Besides the obvious and visible input to the formatter (the document source) you'll need to keep the two hidden inputs of the formatter (DTD and stylesheet) in mind to understand what follows.

Other DTDs

A brief digression into other DTDs may help clarify what parts of the previous section are specific to DocBook and what parts are general to all structural-markup languages.

TEI [http://www.tei-c.org/] (Text Encoding Initiative) is a large, elaborate DTD used primarily in academia for computer transcription of literary texts. TEI's Unix-based toolchains use many of the same tools that are involved with DocBook, but with different stylesheets and (of course) a different DTD.

XHTML, the latest version of HTML, is also an XML application described by a DTD, which explains the family resemblance between XHTML and DocBook tags. The XHTML toolchain consists of Web browsers that can format HTML as flat ASCII, together with any of a number of ad-hoc HTML-to-print utilities.

Many other XML DTDs are maintained to help people exchange structured information in fields as diverse as bioinformatics and banking. You can look at a list of repositories [http://www.xml.com/pub/rg/DTD_Repositories] to get some idea of the variety available.

The DocBook Toolchain

Normally, what you'll do to make XHTML from your DocBook sources is use the xmlto(1) front end. Your commands will look like this:

```
bash$ xmlto xhtml foo.xml
bash$ ls *.html
ar01s02.html ar01s03.html ar01s04.html index.html
```

In this example, you converted an XML-DocBook document named `foo.xml` with three top-level sections into an index page and two parts. Making one big page is just as easy:

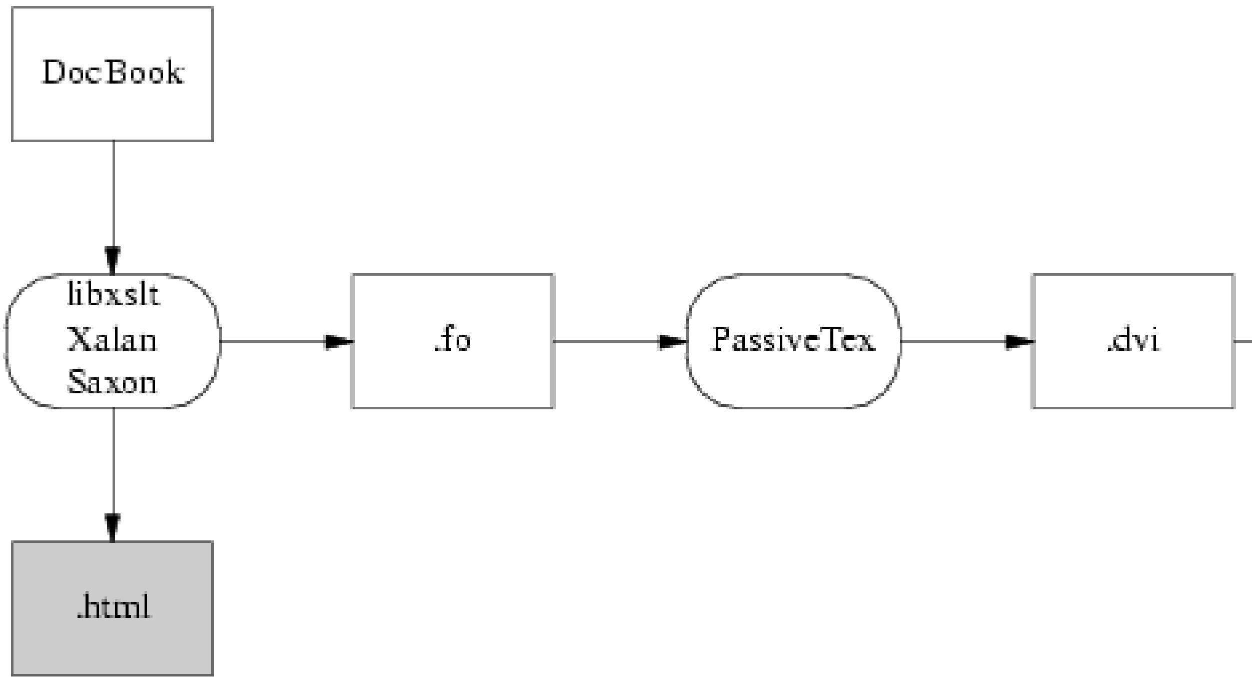
```
bash$ xmlto xhtml-nochunks foo.xml
bash$ ls *.html
foo.html
```

Finally, here is how you make PostScript for printing:

```
bash$ xmlto ps foo.xml      # To make PostScript
bash$ ls *.ps
foo.ps
```

To turn your documents into HTML or PostScript, you need an engine that can apply the combination of DocBook DTD and a suitable stylesheet to your document. Figure 18.2 illustrates how the open-source tools for doing this fit together.

Figure 18.2. Present-day XML-DocBook toolchain.



Parsing your document and applying the stylesheet transformation will be handled by one of three programs. The most likely one is *xsltproc*, the parser that ships with Red Hat Linux. The other possibilities are two Java programs, *Saxon* and *Xalan*.

It is relatively easy to generate high-quality XHTML from either DocBook; the fact that XHTML is simply another XML DTD helps a lot. Translation to HTML is done by applying a rather simple stylesheet, and that's the end of the story. RTF is also simple to generate in this way, and from XHTML or RTF it's easy to generate a flat ASCII text approximation in a pinch.

The awkward case is print. Generating high-quality printed output — which means, in practice, Adobe's PDF (Portable Document Format) — is difficult. Doing it right requires algorithmically duplicating the delicate judgments of a human typesetter moving from content to presentation level.

So, first, a stylesheet translates DocBook's structural markup into another dialect of XML — FO (Formatting Objects). FO markup is very much presentation-level; you can think of it as a sort of XML functional equivalent of *troff*. It has to be translated to PostScript for packaging in a PDF.

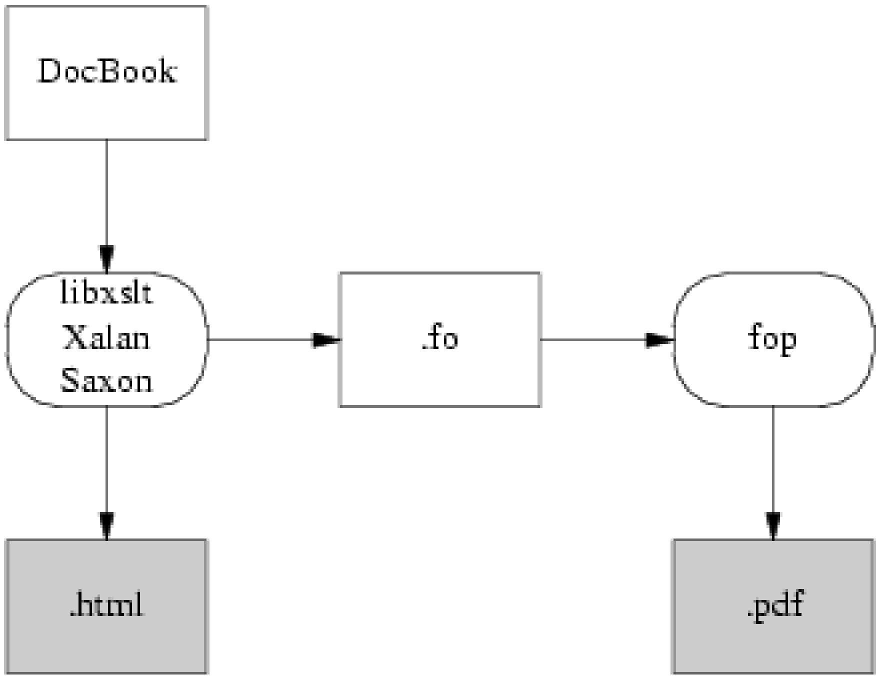
In the toolchain shipped with Red Hat Linux, this job is handled by a *TeX* macro package called *PassiveTeX*. It translates the formatting objects generated by **xsltproc** into Donald Knuth's *TeX* language. *TeX*'s output, known as DVI (DeVice Independent) format, is then massaged into PDF.

If you think this bucket chain of XML to *TeX* macros to DVI to PDF sounds like an awkward kludge, you're right. It clanks, it wheezes, and it has ugly warts. Fonts are a significant problem, since XML and *TeX* and PDF have very different models of how fonts work; also, handling internationalization and localization is a nightmare. About the only thing this code path has going for it is that it works.

The elegant way will be FOP, a direct FO-to-PostScript translator being developed by the Apache project. With FOP, the internationalization problem is, if not solved, at least well confined; XML tools handle Unicode all the way through to FOP. The mapping from Unicode glyphs to Postscript font is also strictly FOP's problem. The only trouble with this approach is that it doesn't work — yet. As of mid-2003, FOP is in an unfinished alpha state — usable, but with rough edges and missing features.

Figure 18.3 illustrates what the FOP toolchain looks like.

Figure 18.3. Future XML-DocBook toolchain with FOP.



FOP has competition. Another project called *xsl-fo-proc* aims to do the same things as FOP, but in C++ (and therefore both faster than Java and not relying on the Java environment). As of mid-2003, *xsl-fo-proc* is in an unfinished alpha state, not as far along as FOP.

Migration Tools

The second biggest problem with DocBook is the effort needed to convert old-style presentation markup to DocBook markup. Human beings can usually parse the presentation of a document into logical structure automatically, because (for example) they can tell from context when an italic font means ‘emphasis’ and when it means something else such as ‘this is a foreign phrase’.

Somehow, in converting documents to DocBook, those sorts of distinctions need to be made explicit. Sometimes they’re present in the old markup; often they are not, and the missing structural information has to be either deduced by clever heuristics or added by a human.

Here is a summary of the state of conversion tools from various other formats. None of these do a completely perfect job; inspection and perhaps a bit of hand-editing by a human being will be needed after conversion.

GNU Texinfo
: The Free Software Foundation intends to support DocBook as an interchange format. Texinfo has enough structure to make reasonably good automatic conversion possible (human editing is still needed afterwards, but not much of it), and the 4.x versions of *makeinfo* feature a `--docbook` switch that generates DocBook. More at the *makeinfo* project page [http://www.gnu.org/directory/texinfo.html].

POD
: A POD::DocBook [http://www.cpan.org/modules/by-module/Pod/] module translates Plain Old Documentation markup to DocBook. It claims to translate every POD tag except the L<> italic tag. The man page also says “Nested =over/=back lists are not supported within DocBook”, but notes that the module has been heavily tested.

LaTeX
: A project called TeX4ht [http://www.lrz-muenchen.de/services/software/sonstiges/tex4h can, according to the author of Passive*TeX*, generate DocBook from *LaTeX*.

man pages and other *troff*-based markups
: These are generally considered the biggest and nastiest conversion problems. And indeed, the basic troff(1) markup is at too low a presentation level for automatic conversion tools to do much of any good. However, the gloom in the picture lightens significantly if we consider translation from sources of documents written in macro packages like man(7). These have enough structural features for automatic translation to get some traction.

: I wrote a tool to do troff-to-DocBook myself, because I couldn’t find anything else that did a tolerable job of it. It’s called doclifter [http://www.catb.org/~esr/doclifter/]. It will translate to either SGML or XML DocBook from man(7), mdoc(7), ms(7), or me(7) macros. See the documentation for details.

Editing Tools

One thing we do not have in mid-2003 is a good open-source structure editor for SGML/XML documents.

LyX [http://www.lyx.org/] is a GUI word processor that uses *LaTeX* for printing and supports structural editing of *LaTeX* markup. There is a *LaTeX* package that generates DocBook, and a how-to document [http://bgu.chez.tiscali.fr/doc/db4lyx/] describing how to write SGML and XML in the LyX GUI.

GNU TeXMacs [http://www.math.u-psud.fr/~anh/TeXmacs/TeXmacs.html] is a project aimed at producing an editor that is good for technical and mathematical material, including displayed formulas. 1.0 was released in April 2002. The developers plan XML support in the future, but it's not there yet.

Most people still hack DocBook tags by hand using either *vi* or *emacs*.

Related Standards and Practices

The tools are coming together, if slowly, to edit and format DocBook markup. But DocBook itself is a means, not an end. We'll need other standards besides DocBook itself to accomplish the searchable-documentation-database objective. There are two big issues: document cataloging and metadata.

The ScrollKeeper [http://scrollkeeper.sourceforge.net/] project aims directly to meet this need. It provides a simple set of script hooks that can be used by package install and uninstall productions to register and unregister their documentation.

ScrollKeeper uses the Open Metadata Format [http://www.ibiblio.org/osrt/omf/]. This is a standard for indexing open-source documentation analogous to a library card-catalog system. The idea is to support rich search facilities that use the card-catalog metadata as well as the source text of the documentation itself.

SGML

In previous sections, we have deliberately omitted a lot of DocBook's history. XML has an older brother, Standard Generalized Markup Language (SGML).

Until mid-2002, no discussion of DocBook would have been complete without a long excursion into SGML, the differences between SGML and XML, and detailed descriptions of the SGML DocBook toolchain. Life can be simpler now; an XML DocBook toolchain is available in open source, works as well as the SGML toolchain ever did, and is easier to use.

XML-DocBook References

One of the things that makes learning DocBook difficult is that the sites related to it tend to overwhelm the newbie with long lists of W3C standards, massive exercises in SGML theology, and dense thickets of abstract terminology. See *XML in a Nutshell* [Harold-Means] for a good book-length general introduction.

Norman Walsh's *DocBook: The Definitive Guide* is available in print [http://www.oreilly.com/catalog/docbook/] and on the Web [http://www.docbook.org/tdg/en/html/docbook.html]. This is indeed the definitive reference, but as an introduction or tutorial it's a disaster. Instead, read this:

Writing Documents Using DocBook [http://xml.web.cern.ch/XML/goossens/dbatcern/]. This is an excellent tutorial.

There is an equally excellent DocBook FAQ [http://www.dpawson.co.uk/docbook/] with a lot of material on styling HTML output. There is also a DocBook wiki [http://docbook.org/wiki/moin.cgi].

Finally, the The XML Cover Pages [http://xml.coverpages.org/] will take you into the jungle of XML standards if you really want to go there.

Best Practices for Writing Unix Documentation

The advice we gave earlier in the chapter about reading Unix documentation can be turned around. When you write documentation for people within the Unix culture, *don't dumb it down*. If you write as if for idiots, you will be written off as an idiot yourself. Dumbing documentation down is very different from making it accessible; the former is lazy and omits important things, whereas the latter requires careful thought and ruthless editing.

Don't think for a moment that volume will be mistaken for quality. And especially, never *ever* omit functional details because you fear they might be confusing, nor warnings about problems because you don't want to look bad. It is *unanticipated* problems that will cost you credibility and users, not the problems you were honest about.

Try to hit a happy medium in information density. Too low is as bad as too high. Use screen shots sparingly; they tend to convey little information beyond the style and feel of the interface. They are never a good substitute for clear textual description.

If your project is of any significant size, you should probably be shipping three different kinds of documentation: man pages as reference material, a tutorial manual, and a FAQ (Frequently Asked Questions) list. You should have a website as well, to serve as a central point of distribution (see the guidelines on communication in Chapter 19).

Huge man pages are viewed with some disfavor; navigation within them can be difficult. If yours are getting large, consider writing a reference manual, with the man page(s) giving a quick summary, pointers into the reference manual, and details of how the program(s) are invoked.

In your source code, include the standard metainformation files described in the Chapter 19 section on open-source release practices, such as `README`. Even if your code is going to be proprietary, these are Unix conventions and future maintainers coming from a Unix background will come up to speed faster if the conventions are followed.

Your man pages should be command references in the traditional Unix style for the traditional Unix audience. The tutorial manual should be long-form documentation for nontechnical users. And the FAQ should be an evolving resource that grows as your software support group learns what the frequent questions are and how to answer them.

There are more specific habits you should adopt if you want to get a little ahead of mid-2003's practice:

1. Maintain your document masters in XML-DocBook. Even your man pages can be DocBook `RefEntry` documents. There is a very good HOWTO [http://www.tldp.org/HOWTO/mini/Man-Page.html] on writing manual pages that explains the sections and organization your users will expect to see.

2. Ship the XML masters. Also, in case your users' systems don't have xmlto(1) ship the *troff* sources that you get by running **xmlto man** on your masters. Your software distribution's installation procedure should install those in the normal way, but direct people to the XML files if they want to write or edit documentation.

3. Make your project's installation package ScrollKeeper-ready.

4. Generate XHTML from your masters (with **xmlto xhtml**) and make it available from your project's Web page.

Whether or not you're using XML-DocBook as a master format, you'll want to find a way to convert your documentation to HTML. Whether your software is open-source or proprietary, users are increasingly likely to find it via the Web. Putting your documentation on-line has the direct effect of making it easier for potential users and customers who know your software exists to read it and learn about it. It has the indirect effect that your software will become more likely to turn up in a Web search.

Chapter 19. Open Source

Programming in the New Unix Community

Software is like sex — it's better when it's free.
--
<author>LinusTorvalds</author>

We concluded Chapter 2 by observing the largest-scale pattern in Unix's history; it flourished when its practices most closely approximated open source, and stagnated when they did not. We then asserted in Chapter 16 that open-source development tools tend to be of high quality. We'll begin this chapter by sketching an explanation of how and why open-source development works. Most of its behaviors are simply intensifications of long-established Unix-tradition practices.

We'll then descend from realm of abstraction and describe some of the most important folk customs that Unix has picked up from the open-source community — in particular, the community-evolved guidelines for what a good source-code release looks like. Many of these customs could be profitably adopted by developers on other modern operating systems as well.

We'll describe these customs on the assumption that you are developing open source; most are still good ideas even if you are writing proprietary software. The open-source assumption is also historically appropriate, because many of these customs found their way back into proprietary Unix shops via ubiquitous open-source tools like patch(1), *Emacs*, and GCC.

Unix and Open Source

Open-source development exploits the fact that characterizing and fixing bugs — unlike, say, implementing a particular algorithm — is a task that lends itself well to being split into multiple parallel subtasks. Exploration of the neighborhood of possibilities near a prototype design also parallelizes well. With the right technological and social machinery in place, development teams that are loosely networked and very large can do astoundingly good work.

Astoundingly, that is, if you are carrying around the mental habits developed by people who treat process secrecy and proprietary control as a given. From *The Mythical Man-Month* [Brooks] until the rise of Linux, the orthodoxy in software engineering was all about small, closely managed teams within heavyweight organizations like corporations and government. The practice was of *large* teams closely managed.

The early Unix community, before the AT&T divestiture, was a paradigmatic example of open source in action. While the pre-divestiture Unix code was technically and legally proprietary, it was treated as a commons within its user/developer community. Volunteer efforts were self-directed by the people most strongly motivated to solve problems. From these choices many good things flowed. Indeed, the technique of open-source development evolved as an unconscious folk practice in the Unix community for more than a quarter century, many years before it was analyzed and labeled in the late 1990s (See *The Cathedral and the Bazaar* [Raymond01] and *Understanding Open Source Software Development* [Feller-Fitzgerald].

In retrospect, it is rather startling how oblivious we all were to the implications of our own behavior. Several people came very close to understanding the phenomenon; Richard Gabriel in his “Worse Is Better” paper from 1990 [Gabriel] is the best known, but one can find prefigurations in Brooks [Brooks] (1975) and as far back as Vyssotsky and Corbató’s meditations on the Multics design (1965). I failed to get it over more than twenty years of observing software development, before being awakened by Linux in the mid-1990s. This experience should make any thoughtful and humble person wonder what other important unifying concepts are still implicit in our behavior and lurking right under our collective noses, hidden not by their complexity but by their very simplicity.

The rules of open-source development are simple:

1. *Let the source be open.* Have no secrets. Make the code and the process that produces it public. Encourage third-party peer review. Make sure that others can modify and redistribute the code freely. Grow the co-developer community as big as you can.

2. *Release early, release often.* A rapid release tempo means quick and effective feedback. When each incremental release is small, changing course in response to real-world feedback is easier.

   Just make sure your first release builds, runs, and demonstrates promise. Usually, an initial version of an open-source program demonstrates promise by doing at least some portion of its final job, sufficient to show that the initiator can actually continue the project. For example, an initial version of a word processor might support typing in text and displaying it on the screen.

   A first release that cannot be compiled or run can kill a project (as, famously, almost happened to the Mozilla browser). Releases that cannot compile suggest that the project developers will be unable to complete the project, Also, non-working programs are difficult for other developers to contribute to, because they cannot easily determine if any change they made improved the program or not.

3. *Reward contribution with praise.* If you can't give your co-developers material rewards, give psychological ones. Even if you can, remember that people will often work harder for reputation than they would for gold.

> A corollary of rule 2 is that individual releases should not be momentous events, with many promises attached and much preparation. It's important to ruthlessly streamline your release process, so that you *can* do frequent releases painlessly. A setup where all other work must stop during release preparation is a terrible mistake. (Notably, if you're using CVS or something similar, releases in preparation should be branches off the main line of development, so that they don't block main-line progress.) To sum up, don't treat releases as big special events; make them part of normal routine.
>
> —
>
> <author>HenrySpencer</author>

Remember that the reason for frequent releases is to shorten and speed the feedback loop connecting your user population to your developers. Therefore, resist thinking of the next release as a polished jewel that cannot ship until everything is perfect. Don't make long wish lists. Make progress incrementally, admit and advertise current bugs, and have confidence that perfection will come with time. Accept that you will go through dozens of point releases on the way, and don't get upset as the version numbers mount.

Open-source development uses large teams of programmers distributed over the Internet and communicating primarily through email and Web documents. Typically, most contributors to any given project are volunteers contributing in order to be rewarded by the increased usefulness of the software to them, and by reputation incentives. A central individual or core group steers the project; other contributors may drop in and drop out sporadically. To encourage casual contributors, it is important to avoid erecting social barriers between them and the core team. Minimize the core team's privileged status, and work hard to keep the boundaries inconspicuous.

Open-source projects follow the Unix-tradition advice of automating wherever possible. They use the patch(1) tool to pass around incremental changes. Many projects (and all large ones) have network-accessible code repositories using version-control systems like CVS (recall the discussion in Chapter 15). Use of automated bug- and patch-tracking systems is also common.

In 1997, almost nobody outside the hacker culture understood that it was even possible to run a large project this way, let alone get high-quality results. In 2003 this is no longer news; projects like Linux, Apache, and Mozilla have achieved both success and high public visibility.

Abandoning the habit of secrecy in favor of process transparency and peer review was the crucial step by which alchemy became chemistry. In the same way, it is beginning to appear that open-source development may signal the long-awaited maturation of software development as a discipline.

Best Practices for Working with Open-Source Developers

Much of what constitutes best practice in the open-source community is a natural adaptation to distributed development; you'll read a lot in the rest of this chapter about behaviors that maintain good communication with other developers. Where Unix conventions are arbitrary (such as the standard names of files that convey metainformation about a source distribution) they often trace back either to Usenet in the early 1980s, or to the conventions and standards of the GNU project.

Good Patching Practice

Most people become involved in open-source software by writing patches for other people's software before releasing projects of their own. Suppose you've written a set of source-code changes for someone else's baseline code. Now put yourself in that person's shoes. How is he to judge whether to include the patch?

It is very difficult to judge the quality of code, so developers tend to evaluate patches by the quality of the submission. They look for clues in the submitter's style and communications behavior instead — indications that the person has been in their shoes and understands what it's like to have to evaluate and merge an incoming patch.

This is actually a rather reliable proxy for code quality. In many years of dealing with patches from many hundreds of strangers, I have only seldom seen a patch that was thoughtfully presented and respectful of my time but technically bogus. On the other hand, experience teaches that patches which look careless or are packaged in a lazy and inconsiderate way are very likely to actually *be* bogus.

Here are some tips on how to get your patch accepted:

Do send patches, don't send whole archives or files.

If your change includes a new file that doesn't exist in the code, then of course you have to send the whole file. But if you're modifying an already-existing file, don't send the whole file. Send a diff

instead; specifically, send the output of the diff(1) command run to compare the baseline distributed version against your modified version.

The diff(1) command and its dual, patch(1), are the most basic tools of open-source development. Diffs are better than whole files because the developer you're sending a patch to may have changed the baseline version since you got your copy. By sending him a diff you save him the effort of separating your changes from his; you show respect for his time.

Send patches against the current version of the code.

It is both counterproductive and rude to send a maintainer patches against the code as it existed several releases ago, and expect him to do all the work of determining which changes duplicate things he has since done, versus which things are actually novel in your patch.

As a patch submitter, it is *your* responsibility to track the state of the source and send the maintainer a minimal patch that expresses what you want done to the main-line codebase. That means sending a patch against the current version.

Don't include patches for generated files.

Before you send your patch, walk through it and delete any patch bands for files in it that are going to be automatically regenerated once the maintainer applies the patch and remakes. The classic examples of this error are C files generated by *Bison* or *Flex*.

These days the most common mistake of this kind is sending a diff with a huge band that is nothing but changebars between your **configure** script and the maintainer's. This file is generated by **autoconf**.

This is inconsiderate. It means your recipient is put to the trouble of separating the real content of the patch from a lot of bulky noise. It's a minor error, not as important as some of the things we'll get to further on — but it will count against you.

Don't send patch bands that just tweak RCS or SCCS $-symbols.

Some people put special tokens in their source files that are expanded by the version-control system when the file is checked in: the Id construct used by RCS and CVS, for example.

If you're using a local version-control system yourself, your changes may alter these tokens. This isn't really harmful, because when your recipient checks his code back in after applying your patch

the tokens will be re-expanded in accordance with the *maintainer's* version-control status. But those extra patch bands are noise. They're distracting. It's more considerate not to send them.

This is another minor error. You'll be forgiven for it if you get the big things right. But you want to avoid it anyway.

Do use `-c` or `-u` format, don't use the default (`-e`) format.

The default (`-e`) format of diff(1) is very brittle. It doesn't include any context, so the patch tool can't cope if any lines have been inserted or deleted in the baseline code since you took the copy you modified.

Getting an `-e` diff is annoying, and suggests that the sender is either an extreme newbie, careless, or clueless. Most such patches get tossed out without a second thought.

Do include documentation with your patch.

This is very important. If your patch makes a user-visible addition or change to the software's features, *include changes to the appropriate man pages and other documentation files in your patch*. Do *not* assume that the recipient will be happy to document your code for you, or to have undocumented features lurking in the code.

Documenting your changes well demonstrates some good things. First, it's considerate to the person you are trying to persuade. Second, it shows that you understand the ramifications of your change well enough to explain it to somebody who can't see the code. Third, it demonstrates that you care about the people who will ultimately use the software.

Good documentation is usually the most visible sign of what separates a solid contribution from a quick and dirty hack. If you take the time and care necessary to produce it, you'll find you're already 85% of the way to having your patch accepted by most developers.

Do include an explanation with your patch.

Your patch should include cover notes explaining why you think the patch is necessary or useful. This is explanation directed not to the users of the software but to the maintainer to whom you are sending the patch.

The note can be short — in fact, some of the most effective cover notes I've ever seen just said "See the documentation updates in this patch". But it should show the right attitude.

The right attitude is helpful, respectful of the maintainer's time, quietly confident but unassuming. It's good to display understanding of the code you're patching. It's good to show that you can identify with the maintainer's problems. It's also good to be up front about any risks you perceive in applying the patch. Here are some examples of the sorts of explanatory comments that experienced developers send:

"I've seen two problems with this code, X and Y. I fixed problem X, but I didn't try addressing problem Y because I don't think I understand the part of the code that I believe is involved".

"Fixed a core dump that can happen when one of the foo inputs is too long. While I was at it, I went looking for similar overflows elsewhere. I found a possible one in blarg.c, near line 666. Are you sure the sender can't generate more than 80 characters per transmission?"

"Have you considered using the Foonly algorithm for this problem? There is a good implementation at <http://www.example.com/~jsmith/foonly.html>".

"This patch solves the immediate problem, but I realize it complicates the memory allocation in an unpleasant way. Works for me, but you should probably test it under heavy load before shipping".

"This may be featuritis, but I'm sending it anyway. Maybe you'll know a cleaner way to implement the feature".

Do include useful comments in your code.

A maintainer will want to have strong confidence that he understands your changes before merging them in. This isn't an invariable rule; if you have a track record of good work with the maintainer, he may just run a casual eye over the changes before checking them in semiautomatically. But everything you can do to help him understand your code and decrease his uncertainty increases your chances that your patch will be accepted.

Good comments in your code help the maintainer understand it. Bad comments don't.

Here's an example of a bad comment:

```
/* norman newbie fixed this 13 Aug 2001 */
```

This conveys no information. It's nothing but a muddy territorial bootprint you're planting in the middle of the maintainer's code. If he takes your patch (which you've made less likely) he will almost certainly strip out this comment. If you want a credit, include a patch band for the project `NEWS` or `HISTORY` file. He's more likely to take that.

Here's an example of a good comment:

```
/*
 * This conditional needs to be guarded so that crunch_data() never
 * gets passed a NULL pointer. <norman_newbie@foosite.com>
 */
```

This comment shows that you understand not only the maintainer's code but the kind of information that he needs to have confidence in your changes. This kind of comment *gives* him confidence in your changes.

Don't take it personally if your patch is rejected

There are lots of reasons a patch can be rejected that don't reflect on you. Remember that most maintainers are under heavy time pressure, and have to be conservative in what they accept lest the project code get broken. Sometime resubmitting with improvements will help. Sometimes it won't. Life is hard.

Good Project- and Archive-Naming Practice

As the load on maintainers of archives like ibiblio, SourceForge, and CPAN increases, there is an increasing trend for submissions to be processed partly or wholly by programs (rather than entirely by a human).

This makes it more important for project and archive-file names to fit regular patterns that computer programs can parse and understand.

Use GNU-style names with a stem and major.minor.patch numbering.

It's helpful to everybody if your archive files all have GNU-like names — all-lower-case alphanumeric stem prefix, followed by a hyphen, followed by a version number, extension, and other suffixes.

A good general form of name has these parts in order:

1. project prefix
2. dash
3. version number
4. dot
5. “src” or “bin” (optional)
6. dot or dash (dot preferred)
7. binary type and options (optional)
8. archiving and compression extensions

Name stems in this style can contain hyphen or underscores to separate syllables; dashes are actually preferred. It is good practice to group related projects by giving the stems a common hyphen-terminated prefix.

Let’s suppose you have a project you call ‘foobar’ at major version 1, minor version or release 2, patchlevel 3. If it’s got just one archive part (presumably the sources), here’s what its names should look like like:

`foobar-1.2.3.tar.gz`	The source archive.
`foobar.lsm`	The LSM file (assuming you’re submitting to ibiblio).

Please *don’t* use names like these:

`foobar123.tar.gz`	This looks to many programs like an archive for a project called “foobar123” with no version number.

`foobar1.2.3.tar.gz` This looks to many programs like an archive for a project called "foobar1" at version 2.3.

`foobar-v1.2.3.tar.gz` Many programs think this goes with a project called "foobar-v1".

`foo_bar-1.2.3.tar.gz` The underscore is hard for people to speak, type, and remember.

`FooBar-1.2.3.tar.gz` Unless you *like* looking like a marketing weenie. This is also hard for people to speak, type, and remember.

If you have to differentiate between source and binary archives, or between different kinds of binary, or express some kind of build option in the file name, please treat that as a file extension to go *after* the version number. That is, please do this:

`foobar-1.2.3.src.tar.gz` Sources.

`foobar-1.2.3.bin.tar.gz` Binaries, type not specified.

`foobar-1.2.3.bin.i386.tar.gz` i386 binaries.

`foobar-1.2.3.bin.i386.static.tar.gz` i386 binaries statically linked.

`foobar-1.2.3.bin.SPARC.tar.gz` SPARC binaries.

Please *don't* use names like 'foobar-i386-1.2.3.tar.gz', because programs have a hard time telling type infixes (like '-i386') from the stem.

The convention for distinguishing major from minor release is simple: you increment the patch level for fixes or minor features, the minor version number for compatible new features, and the major version number when you make incompatible changes.

But respect local conventions where appropriate.

Some projects and communities have well-defined conventions for names and version numbers that aren't necessarily compatible with the above advice. For instance, Apache modules are generally named like mod_foo, and have both their own version number and the version of Apache with which they work. Likewise, Perl modules have version numbers that can be treated as floating point numbers (e.g., you might see 1.303 rather than 1.3.3), and the distributions are generally named

Foo-Bar-1.303.tar.gz for version 1.303 of module Foo::Bar. (Perl itself, on the other hand, switched to using the conventions described here in late 1999.)

Look for and respect the conventions of specialized communities and developers; for general use, follow the above guidelines.

Try hard to choose a name prefix that is unique and easy to type.

The stem prefix should be common to all of a project's files, and it should be easy to read, type, and remember. So please don't use underscores. And don't capitalize or BiCapitalize without extremely good reason — it messes up the natural human-eyeball search order and looks like some marketing weenie trying to be clever.

It confuses people when two different projects have the same stem name. So try to check for collisions before your first release. Two good places to check are the index file of ibiblio [http://metalab.unc.edu/pub/Linux] and the application index at Freshmeat [http://www.freshmeat.net]. Another good place to check is SourceForge [http://www.sourceforge.net]; do a name search there.

Good Development Practice

Here are some of the behaviors that can make the difference between a successful project with lots of contributors and one that stalls out after attracting no interest:

Don't rely on proprietary code.

Don't rely on proprietary languages, libraries, or other code. Doing so is risky business at the best of times; in the open-source community, it is considered downright rude. Open-source developers don't trust code for which they can't review the source.

Use GNU Autotools.

Configuration choices should be made at compile time. A significant advantage of open-source distributions is that they allow the package to adapt at compile-time to the environment it finds. This is critical because it allows the package to run on platforms its developers have never seen, and it allows the software's community of users to do their own ports. Only the largest of development teams can afford to buy all the hardware and hire enough employees to support even a limited number of platforms.

Therefore: Use the GNU autotools to handle portability issues, do system-configuration probes, and tailor your makefiles. People building from sources today expect to be able to type **configure; make; make install** and get a clean build — and rightly so. There is a good tutorial on these tools [http://seul.org/docs/autotut/].

autoconf and *autoheader* are mature. *automake*, as we've previously noted, is still buggy and brittle as of mid-2003; you may have to maintain your own `Makefile.in`. Fortunately it's the least important of the autotools.

Regardless of your approach to configuration, do not ask the user for system information at compile-time. The user installing the package does not know the answers to your questions, and this approach is doomed from the start. The software must be able to determine for itself any information that it may need at compile- or install-time.

But *autoconf* should not be regarded as a license for knob-ridden designs. If at all possible, program to standards like POSIX and refrain also from asking the system for configuration information. Keep ifdefs to a minimum — or, better yet, have none at all.

Test your code before release.

A good test suite allows the team to easily run regression tests before releases. Create a strong, usable test framework so that you can incrementally add tests to your software without having to train developers in the specialized intricacies of the test suite.

Distributing the test suite allows the community of users to test their ports before contributing them back to the group.

Encourage your developers to use a wide variety of platforms as their desktop and test machines, so that code is continuously being tested for portability flaws as part of normal development.

It is good practice, and encourages confidence in your code, when it ships with the test suite you use, and that test suite can be run with **make test**.

Sanity-check your code before release.

By "sanity check" we mean: use every tool available that has a reasonable chance of catching errors a human would be prone to overlook. The more of these you catch with tools, the fewer your users and you will have to contend with.

If you're writing C/C++ using GCC, test-compile with -Wall and clean up all warning messages before each release. Compile your code with every compiler you can find — different compilers often find different problems. Specifically, compile your software on a true 64-bit machine. Underlying datatypes can change on 64-bit machines, and you will often find new problems there. Find a Unix vendor's system and run the lint utility over your software.

Run tools that look for memory leaks and other runtime errors; Electric Fence and Valgrind are two good ones available in open source.

For Python projects, the PyChecker [http://sourceforge.net/projects/pychecker] program can be a useful check. It often catches nontrivial errors.

If you're writing Perl, check your code with perl `-c` (and maybe `-T`, if applicable). Use perl `-w` and 'use strict' religiously. (See the Perl documentation for further discussion.)

Spell-check your documentation and READMEs before release.

Spell-check your documentation, README files and error messages in your software. Sloppy code, code that produces warning messages when compiled, and spelling errors in README files or error messages, all lead users to believe the engineering behind it is also haphazard and sloppy.

Recommended C/C++ Portability Practices

If you are writing C, feel free to use the full ANSI features. Specifically, do use function prototypes, which will help you spot cross-module inconsistencies. The old-style K&R compilers are ancient history.

Do not assume compiler-specific features such as the GCC `-pipe` option or nested functions are available. These will come around and bite you the second somebody ports to a non-Linux, non-GCC system.

Code required for portability should be isolated to a single area and a single set of source files (for example, an `os` subdirectory). Compiler, library and operating system interfaces with portability issues should be abstracted to files in this directory.

A portability layer is a library (or perhaps just a set of macros in header files) that abstracts away just the parts of an operating system's API your program is interested in. Portability layers make it easier to do new software ports. Often, no member of the development team knows the porting platform (for example, there are literally hundreds of different embedded operating systems, and

nobody knows any significant fraction of them). By creating a separate portability layer, it becomes possible for a specialist who knows a platform to port your software without having to understand anything outside the portability layer.

Portability layers also simplify applications. Software rarely needs the full functionality of more complex system calls such as mmap(2) or stat(2), and programmers commonly configure such complex interfaces incorrectly. A portability layer with abstracted interfaces (say, something named `__file_exists` instead of a call to stat(2)) allows you to import only the limited, necessary functionality from the system, simplifying the code in your application.

Always write your portability layer to select based on a feature, never based on a platform. Trying to create a separate portability layer for each supported platform results in a multiple update problem maintenance nightmare. A “platform” is always selected on at least two axes: the compiler and the library/operating system release. In some cases there are three axes, as when Linux vendors select a C library independently of the operating system release. With *M* vendors, *N* compilers, and *O* operating system releases, the number of platforms quickly scales out of reach of any but the largest development teams. On the other hand, by using language and systems standards such as ANSI and POSIX 1003.1, the set of features is relatively constrained.

Portability choices can be made along either lines of code or compiled files. It doesn’t make a difference if you select alternate lines of code on a platform, or one of a few different files. A rule of thumb is to move portability code for different platforms into separate files when the implementations diverge significantly (shared memory mapping on Unix vs. Windows), and leave portability code in a single file when the differences are minimal (for example, whether you’re using `gettimeofday`, `clock_gettime`, `ftime` or `time` to find out the current time-of-day).

For anywhere outside a portability layer, heed this advice:

> `#ifdef` and `#if` are last resorts, usually a sign of failure of imagination, excessive product differentiation, gratuitous “optimization” or accumulated trash. In the middle of code they are anathema. `/usr/include/stdio.h` from GNU is an archetypical horror.
>
> —
>
> <author>DougMcIlroy</author>

Use of `#ifdef` and `#if` is permissible (if well controlled) within a portability layer. Outside it, try hard to confine these to conditionalizing `#includes` based on feature symbols.

Never intrude on the namespace of any other part of the system, including filenames, error return values and function names. Where the namespace is shared, document the portion of the namespace that you use.

Choose a coding standard. The debate over the choice of standard can go on forever — regardless, it is too difficult and expensive to maintain software built using multiple coding standards, and so some common style must be chosen. Enforce your coding standard ruthlessly, as consistency and cleanliness of the code are of the highest priority; the details of the coding standard itself are a distant second.

Good Distribution-Making Practice

These guidelines describe how your distribution should look when someone downloads, retrieves and unpacks it.

Make sure tarballs always unpack into a single new directory.

The single most annoying mistake fledgling contributors make is to build tarballs that unpack the files and directories in the distribution into the current directory, potentially overwriting files already located there. *Never do this!*

Instead, make sure your archive files all have a common directory part named after the project, so they will unpack into a single top-level directory directly *beneath* the current one. Conventionally, the name of the directory should be the same as the stem of the tarball's name. So, for example, a tarball named `foo-0.23.tar.gz` is expected to unpack into a subdirectory named `foo-0.23`.

Example 19.1 shows a makefile trick that, assuming your distribution directory is named “foobar” and SRC contains a list of your distribution files, accomplishes this.

Example 19.1. *tar* archive maker production.

```
foobar-$(VERS).tar.gz:
 @ls $(SRC) | sed s:^:foobar-$(VERS)/: >MANIFEST
 @(cd ..; ln -s foobar foobar-$(VERS))
 (cd ..; tar -czvf foobar/foobar-$(VERS).tar.gz ‘cat foobar/MANIFEST‘)
 @(cd ..; rm foobar-$(VERS))
```

Include a README.

Include a file called `README` that is a roadmap of your source distribution. By ancient convention (originating with Dennis Ritchie himself before 1980, and promulgated on Usenet in the early 1980s), this is the first file intrepid explorers will read after unpacking the source.

`README` files should be short and easy to read. Make yours an introduction, not an epic. Good things to have in the `README` include the following:

1. A brief description of the project.
2. A pointer to the project website (if it has one).
3. Notes on the developer's build environment and potential portability problems.
4. A roadmap describing important files and subdirectories.
5. Either build/installation instructions or a pointer to a file containing same (usually `INSTALL`).
6. Either a maintainers/credits list or a pointer to a file containing same (usually `CREDITS`).
7. Either recent project news or a pointer to a file containing same (usually `NEWS`).
8. Project mailing list addresses.

At one time this file was commonly `READ.ME`, but this interacts badly with browsers, who are all too likely to assume that the `.ME` suffix means it's not textual and can only be downloaded rather than browsed. This usage is deprecated.

Respect and follow standard file-naming practices.

Before even looking at the `README`, your intrepid explorer will have scanned the filenames in the top-level directory of your unpacked distribution. Those names can themselves convey information. By adhering to certain standard naming practices, you can give the explorer valuable clues about where to look next.

Here are some standard top-level file names and what they mean. Not every distribution needs all of these.

README
: The roadmap file, to be read first.

INSTALL
: Configuration, build, and installation instructions.

AUTHORS
: List of project contributors (GNU convention).

NEWS
: Recent project news.

HISTORY
: Project history.

CHANGES
: Log of significant changes between revisions.

COPYING
: Project license terms (GNU convention).

LICENSE
: Project license terms.

FAQ
: Plain-text Frequently-Asked-Questions document for the project.

Note the overall convention that filenames with all-caps names are human-readable metainformation about the package, rather than build components. This elaboration of the `README` was developed early on at the Free Software Foundation.

Having a `FAQ` file can save you a lot of grief. When a question about the project comes up often, put it in the FAQ; then direct users to read the FAQ before sending questions or bug reports. A well-nurtured FAQ can decrease the support burden on the project maintainers by an order of magnitude or more.

Having a `HISTORY` or `NEWS` file with timestamps in it for each release is valuable. Among other things, it may help establish prior art if you are ever hit with a patent-infringement lawsuit (this hasn't happened to anyone yet, but best to be prepared).

Design for upgradability.

Your software will change over time as you put out new releases. Some of these changes will not be backward-compatible. Accordingly, you should give serious thought to designing your installation layouts so that multiple installed versions of your code can coexist on the same system. This is especially important for libraries — you can't count on all your client programs to upgrade in lockstep with your API changes.

The Emacs, Python, and Qt projects have a good convention for handling this: version-numbered directories (another practice that seems to have been made routine by the FSF). Here's how an installed Qt library hierarchy looks (`${ver}` is the version number):

```
/usr/lib/qt
/usr/lib/qt-${ver}
/usr/lib/qt-${ver}/bin       # Where you find moc
/usr/lib/qt-${ver}/lib       # Where you find .so
/usr/lib/qt-${ver}/include     # Where you find header files
```

With this organization, multiple versions can coexist. Client programs have to specify the library version they want, but that's a small price to pay for not having the interfaces break on them. This good practice avoids the notorious "DLL Hell" failure mode of Windows.

Under Linux, provide RPMs.

The de facto standard format for installable binary packages under Linux that used by the Red Hat Package manager, RPM. It's featured in the most popular Linux distribution, and supported by effectively all other Linux distributions (except Debian and Slackware; and Debian can install from RPMs). Accordingly, it's a good idea for your project site to provide installable RPMs as well as source tarballs.

It's also a good idea for you to include in your source tarball the RPM spec file, with a production that makes RPMs from it in your `makefile`. The spec file should have the extension `.spec`; that's how the *rpm* `-t` option finds it in a tarball.

For extra style points, generate your spec file with a shellscript that automatically plugs in the correct version number by analyzing the project `makefile` or a `version.h`.

Note: If you supply source RPMs, use BuildRoot to make the program be built in `/tmp` or `/var/tmp`. If you don't, during the course of running the make install part of your build, the install will install the files in the real final places. This will happen even if there are file collisions, and even if you didn't want to install the package at all. When you're done, the files will have been installed and your system's RPM database will not know about it. Such badly behaved SRPMs are a minefield and should be eschewed.

Provide checksums.

Provide checksums with your binaries (tarballs, RPMs, etc.). This will allow people to verify that they haven't been corrupted or had Trojan-horse code inserted in them.

While there are several commands you can use for this purpose (such as **sum** and **cksum**) it is best to use a cryptographically-secure hash function. The GPG package provides this capability via the `--detach-sign` option; so does the GNU command **md5sum**.

For each binary you ship, your project Web page should list the checksum and the command you used to generate it.

Good Communication Practice

Your software and documentation won't do the world much good if nobody but you knows they exist. Also, developing a visible presence for the project on the Internet will assist you in recruiting users and co-developers. Here are the standard ways to do that.

Announce to Freshmeat.

Announce to Freshmeat [http://www.freshmeat.net]. Besides being widely read itself, this group is a major feeder for Web-based technical news channels.

Never assume the audience has been reading your release announcements since the beginning of time. Always include at least a one-line description of what the software does. Bad example: “Announcing the latest release of FooEditor, now with themes and ten times faster”. Good example: “Announcing the latest release of FooEditor, the scriptable editor for touch-typists, now with themes and ten times faster”.

Announce to a relevant topic newsgroup.

Find a Usenet topic group directly relevant to your application, and announce there as well. Post only where the *function* of the code is relevant, and exercise restraint.

If (for example) you are releasing a program written in Perl that queries IMAP servers, you should certainly post to `comp.mail.imap`. But you should probably not post to `comp.lang.perl` unless the program is also an instructive example of cutting-edge Perl techniques.

Your announcement should include the URL of a project website.

Have a website.

If you intend trying to build any substantial user or developer community around your project, it should have a website. Standard things to have on the website include:

- The project charter (why it exists, who the audience is, etc.).
- Download links for the project sources.
- Instructions on how to join the project mailing list(s).
- A FAQ (Frequently Asked Questions) list.
- HTMLized versions of the project documentation.
- Links to related and/or competing projects.

Refer to the website examples in Chapter 16 for examples of what a well-educated project website looks like.

An easy way to have a website is to put your project on one of the sites that specializes in providing free hosting. In 2003 the two most important of these are SourceForge (which is a demonstration and test site for proprietary collaboration tools) or Savannah (which hosts open-source projects as an ideological statement).

Host project mailing lists.

It's standard practice to have a private development list through which project collaborators can communicate and exchange patches. You may also want to have an announcements list for people who want to be kept informed of the project's progress.

If you are running a project named 'foo', your developer list might be `<foo-dev>` or `<foo-friends>`; your announcement list might be `<foo-announce>`.

An important decision is just how private the "private" development list is. Wider participation in design discussions is often a good thing, but if the list is relatively open, sooner or later you *will* get people asking new-user questions on it. Opinions vary on how best to solve this problem. Just having the documentation tell the new users not to ask elementary questions on the development list is not a solution; such a request must be enforced somehow.

An announcements list needs to be tightly controlled. Traffic should be at most a few messages a month; the whole purpose of such a list is to accommodate people who want to know when something important happens, but don't want to hear about day-to-day details. Most such people will quickly unsubscribe if the list starts generating significant clutter in their mailboxes.

Release to major archives.

See the section *Where Should I Look?* in Chapter 16 for specifics on the major open-source archive sites. You should release your package to these.

Other important locations include:

- The Python Software Activity [http://www.python.org] site (for software written in Python).

- The CPAN [http://language.perl.com/CPAN], the Comprehensive Perl Archive Network (for software written in Perl).

The Logic of Licenses: How to Pick One

The choice of license terms involves decisions about what, if any restrictions the author wants to put on what people do with the software.

If you want to make no restrictions at all, you should put your software in the public domain. An appropriate way to do this would be to include something like the following text at the head of each file:

```
Placed in public domain by J. Random Hacker, 2003.  Share and enjoy!
```

If you do this, you are surrendering your copyright. Anyone can do anything they like with any part of the text. It doesn't get any freer than this.

But very little open-source software is actually placed in the public domain. Some open-source developers want to use their ownership of the code to ensure that it stays open (these tend to adopt the GPL). Others simply want to control their legal exposure; one of the things *all* open-source licenses have in common is a disclaimer of warranty.

Why You Should Use a Standard License

The widely known licenses conforming to the Open Source Definition have well-established interpretive traditions. Developers (and, to the extent they care, users) know what they imply, and have a reasonable take on the risks and tradeoffs they involve. Therefore, use one of the standard licenses carried on the OSI site if at all possible.

If you must write your own license, be sure to have it certified by OSI. This will avoid a lot of argument and overhead. Unless you've been through it, you have no idea how nasty a licensing flamewar can get; people become passionate because the licenses are regarded as almost-sacred covenants touching the core values of the open-source community.

Furthermore, the presence of an established interpretive tradition may prove important if your license is ever tested in court. At time of writing (mid-2003) there is no case law either supporting or invalidating any open-source license. However, it is a legal doctrine (at least in the Unied States, and probably in other common-law countries such as England and the rest of the British Commonwealth) that courts are supposed to interpret licenses and contracts according to the expectations and practices of the community in which they originated. There is thus good reason to hope that open-source community practice will be determinative when the court system finally has to cope.

Varieties of Open-Source Licensing

MIT or X Consortium License

The loosest kind of free-software license is one that grants unrestricted rights to copy, use, modify, and redistribute modified copies as long as a copy of the copyright and license terms is retained in all modified versions. But when you accept this license you do give up the right to sue the maintainers.

You can find a template for the standard X Consortium license at the OSI site [http://www.opensource.org/licenses/mit-license.html].

BSD Classic License

The next least restrictive kind of license grants unrestricted rights to copy, use, modify, and redistribute modified copies as long as a copy of the copyright and license terms is retained in all modified versions, and an acknowledgment is made in advertising or documentation associated with the package. Grantee has to give up the right to sue the maintainers.

The original BSD license is the best-known license of this kind. Among parts of the free-software culture that trace their lineages back to BSD Unix, this license is used even on a lot of free software that was written thousands of miles from Berkeley.

It is also not uncommon to find minor variants of the BSD license that change the copyright holder and omit the advertising requirement (making it effectively equivalent to the MIT license). Note that in mid-1999 the Office of Technology Transfer of the University of California rescinded the advertising clause in the BSD license. So the license on the BSD software has been relaxed in exactly this way. Should you choose the BSD approach, we strongly recommend that you use the new license (without advertising clause) rather than the old. That requirement was dropped because it led to significant legal and procedural complications over what constituted advertising.

You can find a BSD license template at the OSI site [http://www.opensource.org/licenses/bsd-license.html].

Artistic License

The next most restrictive kind of license grants unrestricted rights to copy, use, and locally modify. It allows redistribution of modified binaries, but restricts redistribution of modified sources in ways intended to protect the interests of the authors and the free-software community.

The Artistic License, devised for Perl and widely used in the Perl developer community, is of this kind. It requires modified files to contain “prominent notice” that they have been altered. It also requires people who redistribute changes to make them freely available and make efforts to propagate them back to the free-software community.

You can find a copy of the Artistic License at the OSI site [http://www.opensource.org/licenses/artistic-license.html].

General Public License

The GNU General Public License (and its derivative, the Library or “Lesser” GPL) is the single most widely used free-software license. Like the Artistic License, it allows redistribution of modified sources provided the modified files bear “prominent notice”.

The GPL requires that any program containing parts that are under GPL be wholly GPLed. (The exact circumstances that trigger this requirement are not perfectly clear to everybody.)

These extra requirements actually make the GPL more restrictive than any of the other commonly used licenses. (Larry Wall developed the Artistic License to avoid them while serving many of the same objectives.)

You can find a pointer to the GPL, and instructions about how to apply it, at FSF copyleft site [http://www.gnu.org/copyleft.html].

Mozilla Public License

The Mozilla Public License supports software that is open source, but may be linked with closed-source modules or extensions. It requires that the distributed software ("Covered Code") remain open, but permits add-ons called through a defined API to remain closed.

You can find a template for the MPL at the OSI site [http://www.opensource.org/licenses/MPL-1.1.html].

Chapter 20. Futures

Dangers and Opportunities

The best way to predict the future is to invent it.
--
<author>AlanKay</author>
Uttered during a 1971 meeting at XEROX PARC

History is not over. Unix will continue to grow and change. The community and the tradition around Unix will continue to evolve. Trying to forecast the future is a chancy business, but we can perhaps anticipate it in two ways: first, by looking at how Unix has coped with design challenges in the past; second, by identifying problems that are looking for solutions and opportunities waiting to be exploited.

Essence and Accident in Unix Tradition

To understand how Unix's design might change in the future, we can start by looking at how Unix programming style has changed over time in the past. This effort leads us directly to one of the challenges of understanding the Unix style — distinguishing between accident and essence. That is, recognizing traits that arise from transient technical circumstances versus those that are deeply tied to the central Unix design challenge — how to do modularity and abstraction right while also keeping systems transparent and simple.

This distinction can be difficult, because traits that arose as accidents have sometimes turned out to have essential utility. Consider as an example the 'Silence is golden' rule of Unix interface design we examined in Chapter 11; it began as an adaptation to slow teletypes, but continued because programs with terse output could be combined in scripts more easily. Today, in an environment where having many programs running visibly through a GUI is normal, it has a third kind of utility: silent programs don't distract or waste the user's attention.

On the other hand, some traits that once seemed essential to Unix turned out to be accidents tied to a particular set of cost ratios. For example, old-school Unix favored program designs (and minilanguages like awk(1)) that did line-at-a-time processing of an input stream or record-at-a-time processing of binary files, with any context that needed to be maintained between pieces carried by elaborate state-machine code. New-school Unix design, on the other hand, is generally happy with the assumption that a program can read its entire input into memory and thereafter randomly access

it at will. Indeed, modern Unixes supply an mmap(2) call that allows the programmer to map an entire file into virtual memory and completely hides the serialization of I/O to and from disk space.

This change trades away storage economy to get simpler and more transparent code. It's an adaptation to the plunging cost of memory relative to programmer time. Many of the differences between old-school Unix designs in the 1970s and 1980s and those of the new post-1990 school can be traced to the huge shift in relative costs that today makes all machine resources several orders of magnitude cheaper relative to programmer time than they were in 1969.

Looking back, we can identify three specific technology changes that have driven significant changes in Unix design style: internetworking, bitmapped graphics displays, and the personal computer. In each case, the Unix tradition has adapted to the challenge by discarding accidents that were no longer adaptive and finding new applications for its essential ideas. Biological evolution works this way too. Evolutionary biologists have a rule: “Don't assume that historical origin specifies current utility, or vice versa”. A brief look at how Unix adapted in each of these cases may provide some clues to how Unix might adapt itself to future technology shifts that we cannot yet anticipate.

Chapter 2 described the first of these changes: the rise of internetworking, from the angle of cultural history, telling how TCP/IP brought the original Unix and ARPANET cultures together after 1980. In Chapter 7, the material on obsolescent IPC and networking methods such as System V STREAMS hints at the many false starts, missteps, and dead ends that preoccupied Unix developers through much of the following decade. There was a good deal of confusion about protocols,[153] and about the relationship between intermachine networking and interprocess communication among processes on the same machine.

Eventually the confusion was cleared up when TCP/IP won and BSD sockets reasserted Unix's essential everything-is-a-byte-stream metaphor. It became normal to use BSD sockets for both IPC and networking, older methods for both largely fell out of use, and Unix software grew increasingly indifferent to whether communicating components were hosted on the same or different machines. The invention of the World Wide Web in 1990-1991 was the logical result.

When bitmapped graphics and the example of the Macintosh arrived in 1984 a few years after TCP/IP, they posed a rather more difficult challenge. The original GUIs from Xerox PARC and Apple were beautiful, but wired together far too many levels of the system for Unix programmers

[153]For a few years it looked like ISO's 7-layer networking standard might compete successfully with TCP/IP. It was promoted by a European standards committee politically horrified at the thought of adopting any technology birthed in the bowels of the Pentagon. Alas, their indignation exceeded their technical acuity. The result proved overcomplicated and unhelpful; see [Padlipsky] for details.

to feel comfortable with their design. The prompt response of Unix programmers was to make separation of policy from mechanism an explicit principle; the X windowing system established it by 1988. By splitting X widget sets away from the display manager that was doing low-level graphics, they created an architecture that was modular and clean in Unix terms, and one that could easily evolve better policy over time.

But that was the easy part of the problem. The hard part was deciding whether Unix ought to have a unified interface policy at all, and if so what it ought to be. Several different attempts to establish one through proprietary toolkits (like Motif) failed. Today, in 2003, GTK and Qt contend with each other for the role. While the debate on this question is not over in 2003, the persistence of different UI styles that we noted in Chapter 11 seems telling. New-school Unix design has kept the command line, and dealt with the tension between GUI and CLI approaches by writing lots of CLI-engine/GUI-interface pairs that can be used in both styles.

The personal computer posed few major design challenges as a technology in itself. The 386 and later chips were powerful enough to give the systems designed around them cost ratios similar to those of the minicomputers, workstations, and servers on which Unix matured. The true challenge was a change in the potential market for Unix; the much lower overall price of the hardware made personal computers attractive to a vastly broader, less technically sophisticated user population.

The proprietary-Unix vendors, accustomed to the fatter margins from selling more powerful systems to sophisticated buyers, were never interested in this wider market. The first serious initiatives toward the end-user desktop came out of the open-source community and were mounted for essentially ideological reasons. As of mid-2003, market surveys indicate that Linux has reached about 4%–5% share there, closely comparable to the Apple Macintosh's.

Whether or not Linux ever does substantially better than this, the nature of the Unix community's response is already clear. We examined it in the study of Linux in Chapter 3. It includes adopting a few technologies (such as XML) from elsewhere, and putting a lot of effort into naturalizing GUIs into the Unix world. But underneath the themed GUIs and the installation packaging, the main emphasis is still on modularity and clean code — on getting the infrastructure for serious, high-reliability computing and communications right.

The history of the large-scale desktop-focused developments like Mozilla and OpenOffice.org that were launched in the late 1990s illustrates this emphasis well. In both these cases, the most important theme in community feedback wasn't demand for new features or pressure to make a ship date — it was distaste for monster monoliths, and a general sense that these huge programs would

have to be slimmed down, refactored, and carved into modules before they would be other than embarrassments.

Despite being accompanied by a great deal of innovation, the responses to all three technologies were conservative with regard to the fundamental Unix design rules — modularity, transparency, separation of policy from mechanism, and the other qualities we've tried to characterize earlier in this book. The learned response of Unix programmers, reinforced over thirty years, was to go back to first principles — to try to get more leverage out of Unix's basic abstractions of streams, namespaces, and processes in preference to layering on new ones.

Plan 9: The Way the Future Was

We know what Unix's future used to look like. It was designed by the research group at Bell Labs that built Unix and called 'Plan 9 from Bell Labs'.[154] Plan 9 was an attempt to do Unix over again, better.

The central design challenge the designers attempted to meet in Plan 9 was integrating graphics and ubiquitous networking into a comfortable Unix-like framework. They kept the Unix choice to mediate access to as many system services as possible through a single big file-hierarchy name space. In fact, they improved on it; many facilities that under Unix are accessed through various ad-hoc interfaces like BSD sockets, fcntl(2), and ioctl(2) are in Plan 9 accessed through ordinary read and write operations on special files analogous to device files. For portability and ease of access, almost all device interfaces are textual rather than binary. Most system services (including, for example, the window system) are *file servers* containing special files or directory trees representing the served resources. By representing all resources as files, Plan 9 turns the problem of accessing resources on different servers into the problem of accessing files on different servers.

Plan 9 combined this more-Unix-than-Unix file model with a new concept: private name spaces. Every user (in fact, every process) can have its own view of the system's services by creating its own tree of file-server mounts. Some of the file server mounts will have been manually set up by the user, and others automatically set up at login time. So (as the *Plan 9 from Bell Labs* survey paper points out) "`/dev/cons` always refers to your terminal device and `/bin/date` to the correct version of the date command to run, but which files those names represent depends on circumstances such as the architecture of the machine executing **date**".

[154] The name is a tribute to the 1958 movie that has passed into legend as "the worst ever made", *Plan 9 from Outer Space*. (The legend is, unfortunately, incorrect, as the few who have seen an even worse stinkeroo from 1966 called *Manos: The Hands of Fate* can attest.) Documentation, including a survey paper describing the architecture, along with complete source code and a distribution that installs on PCs, can be readily found with a Web search for the phrase 'Plan 9 from Bell Labs'.

The single most important feature of Plan 9 is that all mounted file servers export the same file-system-like interface, regardless of the implementation behind them. Some might correspond to local file systems, some to remote file systems accessed over a network, some to instances of system servers running in user space (like the window system or an alternate network stack), and some to kernel interfaces. To users and client programs, all these cases look alike.

One of the examples from the Plan 9 survey paper is the way FTP access to remote sites is implemented. There is no ftp(1) command under Plan 9. Instead there is an *ftpfs* fileserver, and each FTP connection looks like a file system mount. *ftpfs* automatically translates open, read, and write commands on files and directories under the mount point into FTP protocol transactions. Thus, all ordinary file-handling tools such as ls(1), mv(1) and cp(1) simply work, both underneath the FTP mount point and across the boundaries with the rest of the user's view of the namespace. The only difference the user (or his scripts and programs) will notice is retrieval speed.

Plan 9 has much else to recommend it, including the reinvention of some of the more problematic areas of the Unix system-call interface, the elimination of superuser, and many other interesting rethinkings. Its pedigree is impeccable, its design elegant, and it exposes some significant errors in the design of Unix. Unlike most efforts at a second system, it produced an architecture that was in many ways simpler and more elegant than its predecessor. Why didn't it take over the world?

One could argue for a lot of specific reasons — lack of any serious effort to market it, scanty documentation, much confusion and stumbling over fees and licensing. For those unfamiliar with Plan 9, it seemed to function mainly as a device for generating interesting papers on operating-systems research. But Unix itself had previously surmounted all these sorts of obstacles to attract a dedicated following that spread it worldwide. Why didn't Plan 9?

The long view of history may tell a different story, but in 2003 it looks like Plan 9 failed simply because it fell short of being a compelling enough improvement on Unix to displace its ancestor. Compared to Plan 9, Unix creaks and clanks and has obvious rust spots, but it gets the job done well enough to hold its position. There is a lesson here for ambitious system architects: the most dangerous enemy of a better solution is an existing codebase that is just good enough.

Some Plan 9 ideas have been absorbed into modern Unixes, particularly the more innovative open-source versions. FreeBSD has a `/proc` file system modeled exactly on that of Plan 9 that can be used to query or control running processes. FreeBSD's rfork(2) and Linux's clone(2) system calls are modeled on Plan 9's rfork(2). Linux's `/proc` file system, in addition to presenting process information, holds a variety of synthesized Plan 9-like device files used to query and control kernel internals using predominantly textual interfaces. Experimental 2003 versions of Linux are

implementing per-process mount points, a long step toward Plan 9's private namespaces. The various open-source Unixes are all moving toward systemwide support for UTF-8, an encoding actually invented for Plan 9.[155]

It may well be that over time, much more of Plan 9 will work its way into Unix as various portions of Unix's architecture slide into senescence. This is one possible line of development for Unix's future.

Problems in the Design of Unix

Plan 9 cleans up Unix, but only really adds one new concept (private namespaces) to its basic set of design ideas. But are there serious problems with those basic design ideas? In Chapter 1 we touched on several issues that Unix arguably got wrong. Now that the open-source movement has put the design future of Unix back in the hands of programmers and technical people, these are no longer decisions we have to live with forever. We'll reexamine them in order to get a better handle on how Unix might evolve in the future.

A Unix File Is Just a Big Bag of Bytes

A Unix file is just a big bag of bytes, with no other attributes. In particular, there is no capability to store information about the file type or a pointer to an associated application program outside the file's actual data.

More generally, everything is a byte stream; even hardware devices are byte streams. This metaphor was a tremendous success of early Unix, and a real advance over a world in which (for example) compiled programs could not produce output that could be fed back to the compiler. Pipes and shell programming sprang from this metaphor.

But Unix's byte-stream metaphor is *so* central that Unix has trouble integrating software objects with operations that don't fit neatly into the byte stream or file repertoire of operations (create, open, read, write, delete). This is especially a problem for GUI objects such as icons, windows, and 'live' documents. Within a classical Unix model of the world, the only way to extend the everything-is-a-byte-stream metaphor is through `ioctl` calls, a notoriously ugly collection of back doors into kernel space.

[155] The tale of how UTF-8 was born involves Ken Thompson, Rob Pike, a new Jersey diner, and a frenzied overnight hack [http://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt].

Fans of the Macintosh family of operating systems tend to be vociferous about this. They advocate a model in which a single filename may have both data and resource 'forks', the data fork corresponding to the Unix byte stream and the resource fork being a collection of name/value pairs. Unix partisans prefer approaches that make file data self-describing so that effectively the same sort of metadata is stored within the file.

The problem with the Unix approach is that every program that writes the file has to know about it. Thus, for example, if we want the file to carry type information inside it, every tool that touches it has to take care to either preserve the type field unaltered or interpret and then rewrite it. While this would be theoretically possible to arrange, in practice it would be far too fragile.

On the other hand, supporting file attributes raises awkward questions about which file operations should preserve them. It's clear that a copy of a named file to another name should copy the source file's attributes as well as its data — but suppose we cat(1) the file, redirecting the output of cat(1) to a new name?

The answer to this question depends on whether the attributes are actually properties of filenames or are in some magical way bundled with the file's data as a sort of invisible preamble or postamble. Then the question becomes: Which operations make the properties visible?

Xerox PARC file-system designs grappled with this problem as far back as the 1970s. They had an 'open serialized' call that returned a byte stream containing both attributes and content. If applied to a directory, it returned a serialization of the directory's attributes plus the serialization of all the files in it. It is not clear that this approach has ever been bettered.

Linux 2.5 already supports attaching arbitrary name/value pairs as properties of a filename, but at time of writing this capability is not yet much used by applications. Recent versions of Solaris have a roughly equivalent feature.

Unix Support for GUIs Is Weak

The Unix experience proves that using a handful of metaphors as the basis for a framework is a powerful strategy (recall the discussion of frameworks and shared context in Chapter 13). The visual metaphor at the heart of modern GUIs (files represented by icons, and opened by clicking which invokes some designated handler program, typically able to create and edit these files) has proven both successful and long-lived, exerting a strong hold on users and interface designers ever since Xerox PARC pioneered it in the 1970s.

Despite considerable recent effort, in 2003 Unix still supports this metaphor only poorly and grudgingly — there are lots of layers, few conventions, and only weak construction utilities. A typical reaction from a Unix old hand is to suspect that this reflects deeper problems with the GUI metaphor itself.

> I think part of the problem is that we still don't have the metaphor right. For example, on the Mac I drag a file to the trashcan to delete it, but when I drag it to a disc it gets copied, and can't drag it to a printer icon to print it because that's done through the menus. I could go on and on. It's like files were in OS/360, before Unix came along with its simple (but not too simple), file idea.
>
> —
>
> <author>SteveJohnson</author>

We quoted Brian Kernighan and Mike Lesk to similar effect in Chapter 11. But the inquiry can't stop with indicting the GUI, because with all its flaws there is tremendous demand for GUIs from end users. Supposing we could get the metaphor right at the level of the design of user interactions, would Unix be capable of supporting it gracefully?

The answer is: probably not. We touched on this problem in considering whether the bag-of-bytes model is adequate. Macintosh-style file attributes may help provide the mechanism for richer support of GUIs, but it seems very unlikely that they are the whole answer. Unix's object model doesn't include the right fundamental constructs. We need to think through what a really strong framework for GUIs would be like — and, just as importantly, how it can be integrated with the existing frameworks of Unix. This is a hard problem, demanding fundamental insights that have yet to emerge from the noise and confusion of ordinary software engineering or academic research.

File Deletion Is Forever

People with VMS experience, or who remember TOPS-20 often miss these systems' file-versioning facilities. Opening an existing file for write or deleting it actually renamed it in a predictable way including a version number; only an explicit removal operation on a version file actually erased data.

Unix does without this, at a not inconsiderable cost in user irritation when the wrong files get deleted through a typo or unexpected effects of shell wildcarding.

There does not seem to be any foreseeable prospect that this will change at the operating system level. Unix developers like clear, simple operations that do what the user tells them to do, even if the user's instructions could amount to commanding "shoot me in the foot". Their instinct is to

say that protecting the user from himself should be done at the GUI or application level, not in the operating system.

Unix Assumes a Static File System

Unix has, in one sense, a very static model of the world. Programs are implicitly assumed to run only briefly, so the background of files and directories can be assumed static during their execution. There is no standard, well-established way to ask the system to notify an application if and when a specified file or directory changes. This becomes a significant issue when writing long-lived user-interface software which wants to know about changes to the background.

Linux has file- and directory-change notification features,[156] and some versions of BSD have copied them, but these are not yet portable to other Unixes.

The Design of Job Control Was Badly Botched

Apart from the ability to suspend processes (in itself a trivial addition to the scheduler which could be made fairly inoffensive) what job control is about is switching a terminal among multiple processes. Unfortunately, it does the easiest part — deciding where keystrokes go — and punts all the hard parts, like saving and restoring the state of the screen, to the application.

A really good implementation of such a facility would be completely invisible to user processes: no dedicated signals, no need to save and restore terminal modes, no need for the applications to redraw the screen at random times. The model ought to be a virtual keyboard that is sometimes connected to the real one (and blocks you if you ask for input when it isn't connected) and a virtual screen which is sometimes visible on the real one (and might or might not block on output when it's not), with the system doing the multiplexing in the same way it multiplexes access to the disk, the processor, etc... and no impact on user programs at all.[157]

Doing it right would have required the Unix tty driver to track the entire current screen state rather than just maintaining a line buffer, and to know about terminal types at kernel level (possibly with help from a daemon process) so it could do restores properly when a suspended process is foregrounded again. A consequence of doing it wrong is that the Unix kernel can't detach a

[156] Look for `F_NOTIFY` under fcntl(2).

[157] This paragraph is based on a 1984 analysis by Henry Spencer. He went on to note that job control was necessary and appropriate for POSIX.1 and later Unix standards to consider precisely *because* it oozes its way into every program, and hence has to be thought about in any application-to-system interface. Hence, POSIX's endorsement of a mis-design, while proper solutions were "out of scope" and hence were not even considered.

session, such as an *xterm* or *Emacs* job, from one terminal and re-attach it to another (which could be of a different type).

As Unix usage has shifted to X displays and terminal emulators, job control has become relatively less important, and this issue does not have quite the force it once did. It is still annoying that there is no suspend/attach/detach, however; this feature could be useful for saving the state of terminal sessions between logins.

A common open-source program called screen(1) solves several of these problems.[158] However, since it has to be called explicitly by the user, its facilities are not guaranteed to be present in every terminal session; also, the kernel-level code that overlaps with it in function has not been removed.

The Unix API Doesn't Use Exceptions

C lacks a facility for throwing named exceptions with attached data.[159] Thus, the C functions in the Unix API indicate errors by returning a distinguished value (usually -1 or a NULL character pointer) and setting a global errno variable.

In retrospect, this is the source of many subtle errors. Programmers in a hurry often neglect to check return values. Because no exception is thrown, the Rule of Repair is violated; program flow continues until the error condition manifests as some kind of failure or data corruption later in execution.

The absence of exceptions also means that some tasks which ought to be simple idioms — like aborting from a signal handler on a version with Berkeley-style signals — have to be performed with code that is complex, subject to portability glitches, and bug-prone.

This problem can be (and normally is) hidden by bindings of the Unix API in languages such as Python or Java that have exceptions.

The lack of exceptions is actually an indicator of a problem with larger immediate implications; C's weak type ontology makes communication between higher-level languages implemented in it problematic. Most of the more modern languages, for example, have lists and dictionaries as

[158] There is a project site for screen(1) at http://www.math.fu-berlin.de/~guckes/screen/.

[159] For nonprogrammers, *throwing an exception* is a way for a program to bail out in the middle of a procedure. It's not quite an exit because the throw can be intercepted by catcher code in an enclosing procedure. Exceptions are normally used to signal errors or unexpected conditions that mean it would be pointless to try to continue normal processing.

primary data types — but, because these don't have any canonical representation in the universe of C, attempting to pass lists between (say) Perl and Python is an unnatural act requiring a lot of glue.

There are technologies that address the larger problem, such as CORBA, but they tend to involve a lot of runtime translation and be unpleasantly heavyweight.

ioctl(2) and fcntl(2) Are an Embarrassment

The ioctl(2) and fcntl(2) mechanisms provide a way to write hooks into a device driver. The original, historical use of ioctl(2) was to set parameters like baud rate and number of framing bits in a serial-communications driver, thus the name (for 'I/O control'). Later, ioctl calls were added for other driver functions, and fcntl(2) was added as a hook into the file system.

Over the years, `ioctl` and `fcntl` calls have proliferated. They are often poorly documented, and often a source of portability problems as well. With each one comes a grubby pile of macro definitions describing operation types and special argument values.

The underlying problem is the same as 'big bag of bytes'; Unix's object model is weak, leaving no natural places to put many auxiliary operations. Designers have an untidy choice among unsatisfactory alternatives; `fcntl`/`ioctl` going through devices in `/dev`, new special-purpose system calls, or hooks through special-purpose virtual file systems that hook into the kernel (e.g. `/proc` under Linux and elsewhere).

It is not clear whether or how Unix's object model will be enriched in the future. If MacOS-like file attributes become a common feature of Unix, tweaking magic named attributes on device drivers may take over the role `ioctl`/`fcntl` now have (this would at least have the merit of not requiring piles of macro definitions before the interface could be used). We've already seen that Plan 9, which uses the named file server or file system as its basic object, rather than the file/bytestream, presents another possible path.

The Unix Security Model May Be Too Primitive

Perhaps root is too powerful, and Unix should have finer-grained capabilities or ACLs (Access Control Lists) for system-administration functions, rather than one superuser that can do anything. People who take this position argue that too many system programs have permanent root privileges through the set-user-ID mechanism; if even one can be compromised, intrusions everywhere will follow.

This argument is weak, however. Modern Unixes allow any given user account to belong to multiple security groups. Through use of the execute-permission and set-group-ID bits on program executables, each group can in effect function as an ACL for files or programs.

This theoretical possibility is very little used, however, suggesting that the demand for ACLs is much less in practice than it is in theory.

Unix Has Too Many Different Kinds of Names

Unix unified files and local devices — they're all just byte streams. But network devices accessed through sockets have different semantics in a different namespace. Plan 9 demonstrates that files can be smoothly unified with both local and remote (network) devices, and all of these things can be managed through a namespace that is dynamically adjustable per-user and even per-program.

File Systems Might Be Considered Harmful

Was having a file system at all the wrong thing? Since the late 1970s there has been an intriguing history of research into persistent object stores and operating systems that don't have a shared global file system at all, but rather treat disk storage as a huge swap area and do everything through virtualized object pointers.

Modern efforts in this line (such as EROS[160]) hint that such designs can offer large benefits including both provable conformance to a security policy and higher performance. It must be noted, however, that if this is a failure of Unix, it is equally a failure of all of its competitors; no major production operating system has yet followed EROS's lead.[161]

Towards a Global Internet Address Space

Perhaps URLs don't go far enough. We'll leave the last word on possible future directions of Unix to Unix's inventor:

> My ideal for the future is to develop a file system remote interface (a la Plan 9) and then have it implemented across the Internet as the standard rather than HTML. That would be ultimate cool.
>
> —

[160]http://www.eros-os.org/

[161]The operating systems of the Apple Newton, the AS/400 minicomputer and the Palm handheld could be considered exceptions.

<author>KenThompson</author>

Problems in the Environment of Unix

The old-time Unix culture has largely reinvented itself in the open-source movement. Doing so saved us from extinction, but it also means that the problems of open source are now ours as well.

One of these is how to make open-source development economically sustainable. We have reconnected with our roots in the collaborative, open process of Unix's early days. We have largely won the technical argument for abandoning secrecy and proprietary control. We have thought of ways to cooperate with markets and managers on more equal terms than we ever could in the 1970s and 1980s, and in many ways our experiments have succeeded. In 2003 the open-source Unixes, and their core development groups, have achieved a degree of mainstream respectability and authority that would have been unimaginable as recently as the mid-1990s.

We have come a long way. But we have a long way to go yet. We know what business models might work in theory, and now we can even point at a sporadic handful of successes that demonstrate that they work in practice; now we have to show that they can be made to work reliably over a longer term.

It's not necessarily going to be an easy transition. Open source turns software into a service industry. Service-provider firms (think of medical and legal practices) can't be scaled up by injecting more capital into them; those that try only scale up their fixed costs, overshoot their revenue base, and starve to death. The choices come down to singing for your supper (getting paid through tips and donations), running a corner shop (a small, low-overhead service business), or finding a wealthy patron (some large firm that needs to use and modify open-source software for its business purposes).

In total, the amount of money spent to hire software developers can be expected to rise, for the same reasons that mechanics' hourly wages go up as the price of automobiles drops.[162] But it is going to become more difficult for any one individual or firm to capture a lot of that spending. There will be many programmers who are well off, but fewer millionaires. This is actually a sign of progress, of inefficiencies being competed out of the system. But it will represent a big change in climate, and probably means that investors will lose what little interest they have left in funding software startups.

[162] For a more complete discussion of this effect, see *The Magic Cauldron* in [Raymond01].

One important subproblem related to the increasing difficulty of sustaining really large software businesses is how to organize end-user testing. Historically, the Unix culture's concentration on infrastructure has meant that we have not tended to build programs that depended for their value on providing a comfortable interface for end-users. Now, especially in the open-source Unixes that aim to compete directly with Microsoft and Apple, that is changing. But end-user interfaces need to be systematically tested with real end users — and therein lie some challenges.

Real end-user testing demands facilities, specialists, and a level of monitoring that are difficult for the distributed volunteer groups characteristic of open-source development to arrange. It may be, therefore, that open-source word processors, spreadsheets, and other 'productivity' applications have to be left in the hands of large corporate-sponsored efforts like OpenOffice.org that can afford the overhead. Open-source developers consider single corporations to be single points of failure and worry about such dependencies, but no better solution has yet evolved.

These are economic problems. We have other problems of a more political nature, because success makes enemies.

Some are familiar. Microsoft's ambition for an unchallengeable monopoly lock on computing made the defeat of Unix a strategic goal for the company in the mid-1980s, five years before we knew we were in a fight. In mid-2003, despite having had several growth markets it was counting on largely usurped by Linux, Microsoft is still the wealthiest and most powerful software company in the world. Microsoft knows very well that it must defeat the new-school Unixes of the open-source movement to survive. To defeat them, it must destroy or discredit the culture that produced them.

Unix's comeback in the hands of the open-source community, and its association with the free-wheeling culture of the Internet, has made it newer enemies as well. Hollywood and Big Media feel deeply threatened by the Internet and have launched a multipronged attack on uncontrolled software development. Existing legislation like the Digital Millennium Copyright Act has already been used to prosecute software developers who were doing things the media moguls disliked (the most notorious cases, of course, involve the DeCSS software that enables copying of encypted DVDs). Contemplated schemes like the so-called Trusted Computing Platform Alliance and Palladium threaten[163] to make open-source development effectively illegal — and if open source goes down, Unix is very likely to go down with it.

Unix and the hackers and the Internet against Microsoft and Hollywood and Big Media. It's a struggle we need to win for all our traditional reasons of professionalism, allegiance to our craft,

[163]See the TCPA FAQ [http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html] for a rather hair-raising summary of the possibilities by a noted security specialist.

and mutual tribal loyalty. But there are larger reasons this struggle is important. The possibilities of politics are increasingly shaped by communication technology — who can use it, who can censor it, who can control it. Government and corporate control of the content of the nets, and of what people can do with their computers, is a severe long-term threat to political freedom. The nightmare scenario is one in which corporate monopolism and statist power-seeking, always natural allies, feed back into each other and create rationales for increasing regulation, repression, and criminalization of digital speech. In opposing this, we are the warriors of liberty — not merely our own liberty, but everyone else's as well.

Problems in the Culture of Unix

Just as important as the technical problems with Unix itself and the challenges consequent on its success are the cultural problems of the community around it. There are at least two serious ones: a lesser challenge of internal transition, and a greater one of overcoming our historical elitism.

The lesser challenge is that of friction between the old-school Unix gurus and the new-school open-source crowd. The success of Linux, in particular, is not an entirely comfortable phenomenon for a lot of older Unix programmers. This is partly a generational problem. The raucous energy, naïvete and gleeful zealotry of the Linux kids sometimes grates on elders who have been around since the 1970s and (often rightly) consider themselves wiser. It's only exacerbated by the fact that the kids are succeeding where the elders failed.

The greater problem of psychology only became clear to me after spending three days at a Macintosh developer conference in 2000. It was a very enlightening experience to be immersed in a programming culture with assumptions diametrically opposed to those of the Unix world.

Macintosh programmers are all about the user experience. They're architects and decorators. They design from the outside in, asking first "What kind of interaction do we want to support?" and then building the application logic behind it to meet the demands of the user-interface design. This leads to programs that are very pretty and infrastructure that is weak and rickety. In one notorious example, as late as Release 9 the MacOS memory manager sometimes required the user to manually deallocate memory by manually chucking out exited but still-resident programs. Unix people are viscerally revolted by this kind of mal-design; they don't understand how Macintosh people could live with it.

By contrast, Unix people are all about infrastructure. We are plumbers and stonemasons. We design from the inside out, building mighty engines to solve abstractly defined problems (like "How do we get reliable packet-stream delivery from point A to point B over unreliable hardware and

links?"). We then wrap thin and often profoundly ugly interfaces around the engines. The commands date(1), find(1), and ed(1) are notorious examples, but there are hundreds of others. Macintosh people are viscerally revolted by this kind of mal-design; they don't understand how Unix people can live with it.

Both design philosophies have some validity, but the two camps have a great deal of difficulty seeing each other's points. The typical Unix developer's reflex is to dismiss Macintosh software as gaudy fluff, eye-candy for the ignorant, and to continue building software that appeals to other Unix developers. If end-users don't like it, so much the worse for the end users; they will come around when they get a clue.

In many ways this kind of parochialism has served us well. We are the keepers of the Internet and the World Wide Web. Our software and our traditions dominate serious computing, the applications where 24/7 reliability and minimal downtime is a must. We really are extremely good at building solid infrastructure; not perfect by any means, but there is no other software technical culture that has anywhere close to our track record, and it is one to be proud of.

The problem is that we increasingly face challenges that demand a more inclusive view. Most of the computers in the world don't live in server rooms, but rather in the hands of those end users. In early Unix days, before personal computers, our culture defined itself partly as a revolt against the priesthood of the mainframes, the keepers of the big iron. Later, we absorbed the power-to-the-people idealism of the early microcomputer enthusiasts. But today *we* are the priesthood; *we* are the people who run the networks and the big iron. And our implicit demand is that if you want to use our software, you must learn to think like us.

In 2003, there is a deep ambivalence in our attitude — a tension between elitism and missionary populism. We want to reach and convert the 92% of the world for whom computing means games and multimedia and glossy GUI interfaces and (at their most technical) light email and word processing and spreadsheets. We are spending major effort on projects like GNOME and KDE designed to give Unix a pretty face. But we are still elitists at heart, deeply reluctant and in many cases unable to identify with or listen to the needs of the Aunt Tillies of the world.

To non-technical end users, the software we build tends to be either bewildering and incomprehensible, or clumsy and condescending, or both at the same time. Even when we try to do the user-friendliness thing as earnestly as possible, we're woefully inconsistent at it. Many of the attitudes and reflexes we've inherited from old-school Unix are just wrong for the job. Even when we want to listen to and help Aunt Tillie, we don't know how — we project our categories and our concerns onto her and give her 'solutions' that she finds as daunting as her problems.

Our greatest challenge as a culture is whether we can outgrow the assumptions that have served us so well — whether we can acknowledge, not merely intellectually but in the sinew of daily practice, that the Macintosh people have a point. Their point is made in more general, less Mac-specific way in *The Inmates Are Running the Asylum* [Cooper], an insightful and argumentative book about what its author calls *interaction design* that (despite occasional crotchets) contains a good deal of hard truth that every Unix programmer ought to know.

We can turn aside from this; we can remain a priesthood appealing to a select minority of the best and brightest, a geek meritocracy focused on our historical role as the keepers of the software infrastructure and the networks. But if we do this, we will very likely go into decline and eventually lose the dynamism that has sustained us through decades. Someone else will serve the people; someone else will put themselves where the power and the money are, and own the future of 92% of all software. The odds are, whether that someone else is Microsoft or not, that they will do it using practices and software we don't much like.

Or we can truly accept the challenge. The open-source movement is trying hard to do so. But the kind of sustained work and intelligence we have brought to other problems in the past will not alone suffice. Our attitudes must change in a fundamental and difficult way.

In Chapter 4 we discussed the importance of throwing away limiting assumptions and discarding the past in solving technical problems, suggesting a parallel with the Zen ideas of detachment and 'beginner's mind'. We have a larger kind of detachment to work on now. We must learn humility before Aunt Tillie, and relinquish some of the long-held prejudices that have made us so successful in the past.

Tellingly, the Macintosh culture has begun to converge with ours — MacOS X has Unix underneath, and in 2003 Mac developers are (albeit with a struggle in some cases) making the mental adjustment to learn the infrastructure-focused virtues of Unix. Our challenge will be, reciprocally, to embrace the user-centered virtues of the Macintosh.

There are other signs that the Unix culture is shedding its insularity as well. One is the convergence that seems to be going on between the Unix/open-source community and the movement called "agile programming".[164] We noted in Chapter 4 that Unix programmers have seized happily on the concept of refactoring, one of the preoccupations of the agile-programming thinkers. Refactoring, and other agile concepts like unit-testing and design around stories, seem to articulate and sharpen practices that have heretofore been widespread but only implicit in the Unix tradition. The Unix tradition, on

[164] For an introduction to agile programming, see the Agile Manifesto [http://agilemanifesto.org/]

the other hand, can bring groundedness and the lessons of long experience to the agile-programming party. As open-source software gains market share it is even conceivable that these cultures will fuse, much as the old-time Internet and early Unix cultures did after 1980.

Reasons to Believe

The future of Unix is full of difficult problems. Would we truly want it any other way?

For more than thirty years we have thrived on challenges. We pioneered the best practices of software engineering. We created today's Internet and Web. We have built the largest, most complex, and most reliable software systems ever to exist. We outlasted the IBM monopoly and we're making a run against the Microsoft monopoly that is good enough to deeply frighten it.

Not that everything has been triumph by any means. In the 1980s we nearly destroyed ourselves by acceding to the proprietary capture of Unix. We neglected the low end, the nontechnical end users, for far too long and thereby left Microsoft an opening to grossly lower the quality standards of software. Intelligent observers have pronounced our technology, our community, and our values to be dead any number of times.

But always we have come storming back. We make mistakes, but we learn from our mistakes. We have transmitted our culture across generations; we have absorbed much of what was best from the early academic hackers and the ARPANET experimenters and the microcomputer enthusiasts and a number of other cultures. The open-source movement has resurrected the vigor and idealism of our early years, and today we are stronger and more numerous than we have ever been.

So far, betting against the Unix hackers has always been short-term smart but long-term stupid. We can prevail — if we choose to.

Appendix A. Glossary of Abbreviations

The most important abbreviations and acronyms used in the main text are defined here.

API — *Application Programming Interface.* The set of procedure calls that communicates with a linkable procedure library or an operating-system kernel or the combination of both.

BSD — *Berkeley System Distribution*; also *Berkeley Software Distribution*; sources are ambiguous. The generic name of the Unix distributions issued by the Computer Science Research Group at the University of California at Berkeley between 1976 and 1994, and of the open-source Unixes genetically descended from them.

CLI — *Command Line Interface.* Considered archaic by some, but still very useful in the Unix world.

CPAN — *Comprehensive Perl Archive Network.* The central Web repository [http://cpan.org/] for Perl modules and extensions.

GNU — *GNU's Not Unix!* The recursive acronym for the Free Software Foundation's project to produce an entire free-software clone of Unix. This effort didn't completely succeed, but did produce many of the essential tools of modern Unix development including Emacs and the GNU Compiler Collection.

GUI — *Graphical User Interface.* The modern style of application interface using mice, windows, and icons invented at Xerox PARC during the 1970s, as opposed to the older CLI or roguelike styles.

IDE — *Integrated Development Environment.* A GUI workbench for developing code, featuring facilities like symbolic debugging, version control, and data-structure browsing. These are not commonly used under Unix, for reasons discussed in Chapter 15.

IETF — *Internet Engineering Task Force.* The entity responsible for defining Internet protocols such as TCP/IP. A loose, collegial organization mainly of technical people.

IPC — *Inter-Process Communication.* Any method of passing data between processes running in separate address spaces.

MIME — *Multipurpose Internet Mail Extensions.* A series of RFCs that describe standards for embedding binary and multipart messages within RFC-822 mail. Besides being used for mail transport, MIME is used as an underlevel by important application protocols including HTTP and BEEP.

OO — *Object Oriented.* A style of programming that tries to encapsulate data to be manipulated and the code that manipulates it in (theoretically) sealed containers called objects. By contrast, non-object-oriented programming is more casual about exposing the internals of the data structure and code.

OS — *Operating System.* The foundation software of a machine; that which schedules tasks, allocates storage, and presents a default interface to the user between applications. The facilities an operating system provides and its general design philosophy exert an extremely strong influence on programming style and on the technical cultures that grow up around its host machines.

PDF — *Portable Document Format.* The PostScript language for control of printers and other imaging devices is designed to be streamed to printers. PDF is a sequence of PostScript pages, packaged with annotations so it can conveniently be used as a display format.

PDP-11 — *Programmable Data Processor 11.* Possibly the single most successful minicomputer design in history; first shipped in 1970, last shipped in 1990, and the immediate ancestor of the VAX. The PDP-11 was the first major Unix platform.

PNG — *Portable Network Graphics.* The World Wide Web Consortium's standard and recommended format for bitmap graphics images. An elegantly designed binary graphics format described in Chapter 5.

RFC — *Request For Comment.* An Internet standard. The name arose at a time when the documents were regarded as proposals to be submitted to a then-nonexistent but anticipated formal approval process of some sort. The formal approval process never materialized.

RPC — *Remote Procedure Call.* Use of IPC methods that attempt to create the illusion that the processes exchanging them are running in the same address space, so they can cheaply (a) share complex structures, and (b) call each other like function libraries, ignoring latency and other performance considerations. This illusion is notoriously difficult to sustain.

TCP/IP — *Transmission Control Protocol/Internet Protocol.* The basic protocol of the Internet since the conversion from NCP (Network Control Protocol) in 1983. Provides reliable transport of data streams.

UDP/IP — *Universal Datagram Protocol/Internet Protocol.* Provides unreliable but low-latency transport for small data packets.

UI *User Interface.*

VAX Formally, *Virtual Address Extension*: the name of a classic minicomputer design developed by Digital Equipment Corporation (later merged with Compaq, later merged with Hewlett-Packard) from the PDP-11. The first VAX shipped in 1977. For ten years after 1980 VAXen were among the most important Unix platforms. Microprocessor reimplementations are still shipping today.

Appendix B. References

Event timelines of the Unix Industry [http://snap.nlc.dcccd.edu/learn/drkelly/hst-hand.htm] and of GNU/Linux and Unix [http://www.robotwisdom.com/linux/timeline.html] are available on the Web. A timeline tree of Unix releases [http://www.levenez.com/unix/] is also available.

Bibliography

[Appleton] Randy Appleton. *Improving Context Switching Performance of Idle Tasks under Linux*. 2001.

Available on the Web [http://cs.nmu.edu/~randy/Research/Papers/Scheduler/].

[Baldwin-Clark] Carliss Baldwin and Kim Clark. *Design Rules, Vol 1: The Power of Modularity*. 2000. MIT Press. ISBN 0-262-024667.

[Bentley] Jon Bentley. *Programming Pearls*. 2nd Edition. 2000. Addison-Wesley. ISBN 0-201-65788-0.

The third essay in this book, “Data Structures Programs”, argues a case similar to that of Chapter 9 with Bentley’s characteristic eloquence. Some of the book is available on the Web [http://www.cs.bell-labs.com/cm/cs/pearls/].

[BlaauwBrooks] Gerrit A. Blaauw and Frederick P. Brooks. *Computer Architecture: Concepts and Evolution*. 1997. ISBN 0-201-10557-8. Addison-Wesley.

[Bolinger-Bronson] Dan Bolinger and Tan Bronson. *Applying RCS and SCCS*. O’Reilly & Associates. 1995. ISBN 1-56592-117-8.

Not just a cookbook, this also surveys the design issues in version-control systems.

[Brokken] Frank Brokken. *C++ Annotations Version*. 2002.

Available on the Web [http://www.icce.rug.nl/documents/cplusplus/cplusplus.html].

[BrooksD] David Brooks. *Converting a UNIX .COM Site to Windows*. 2000.

Available on the Web [http://www.securityoffice.net/mssecrets/hotmail.html#_Toc491601819].

[Brooks] Frederick P. Brooks. *The Mythical Man-Month*. 20th Anniversary Edition. Addison-Wesley. 1995. ISBN 0-201-83595-9.

[Boehm] Hans Boehm. *Advantages and Disadvantages of Conservative Garbage Collection*.

Thorough discussion of tradeoffs between garbage-collected and non-garbage-collected environments. Available on the Web [http://www.hpl.hp.com/personal/Hans_Boehm/gc/issues.html].

[Cameron] Debra Cameron, Bill Rosenblatt, and Eric Raymond. *Learning GNU Emacs*. 2nd Edition. O'Reilly & Associates. 1996. ISBN 1-56592-152-6.

[Cannon] L. W. Cannon, R. A. Elliot, L. W. Kirchhoff, J. A. Miller, J. M. Milner, R. W. Mitzw, E. P. Schan, N. O. Whittington, Henry Spencer, David Keppel, and Mark Brader. *Recommended C Style and Coding Standards*. 1990.

An updated version of the *Indian Hill C Style and Coding Standards* paper, with modifications by the last three authors. It describes a recommended coding standard for C programs. Available on the Web [http://www.apocalypse.org/pub/u/paul/docs/cstyle/cstyle.htm].

[Christensen] Clayton Christensen. *The Innovator's Dilemma*. HarperBusiness. 2000. ISBN 0-066-62069-4.

The book that introduced the term "disruptive technology". A fascinating and lucid examination of how and why technology companies doing everything right get mugged by upstarts. A business book technical people should read.

[Comer] *Unix Review*. Douglas Comer. "Pervasive Unix: Cause for Celebration". October 1985. p. 42.

[Cooper] Alan Cooper. *The Inmates Are Running the Asylum*. Sams. 1999. ISBN 0-672-31649-8.

Despite some occasional quirks and crotchets, this book is a trenchant and brilliant analysis of what's wrong with software interface designs, and how to put it right.

[Coram-Lee] Tod Coram and Ji Lee. *Experiences — A Pattern Language for User Interface Design*. 1996.

Available on the Web [http://www.maplefish.com/todd/papers/Experiences.html].

[DuBois] Paul DuBois. *Software Portability with Imake*. O'Reilly & Associates. 1993. ISBN 1-56592-055-4.

[Eckel] Bruce Eckel. *Thinking in Java*. 3rd Edition. Prentice-Hall. 2003. ISBN 0-13-100287-2.

Available on the Web [http://www.mindview.net/Books/TIJ/].

[Feller-Fitzgerald] Joseph Feller and Brian Fitzgerald. *Understanding Open Source Software*. 2002. ISBN 0-201-73496-6. Addison-Wesley.

[FlanaganJava] David Flanagan. *Java in a Nutshell*. O'Reilly & Associates. 1997. ISBN 1-56592-262-X.

[FlanaganJavaScript] David Flanagan. *JavaScript: The Definitive Guide*. 4th Edition. O'Reilly & Associates. 2002. ISBN 1-596-00048-0.

[Fowler] Martin Fowler. *Refactoring*. Addison-Wesley. 1999. ISBN 0-201-48567-2.

[Friedl] Jeffrey Friedl. *Mastering Regular Expressions*. 2nd Edition. 2002. ISBN 0-596-00289-0. O'Reilly & Associates. 484pp..

[Fuzz] Barton Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. *Fuzz Revisited: A Re-examination of the Reliability of Unix Utilities and Services*. 2000.

Available on the Web [http://www.opensource.org/advocacy/fuzz-revisited.pdf].

[Gabriel] Richard Gabriel. *Good News, Bad News, and How to Win Big*. 1990.

Available on the Web [http://www.dreamsongs.com/WorseIsBetter.html].

[Gancarz] Mike Gancarz. *The Unix Philosophy*. Digital Press. 1995. ISBN 1-55558-123-4.

[GangOfFour] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1997. ISBN 0-201-63361-2.

[Garfinkel] Simson Garfinkel, Daniel Weise, and Steve Strassman. *The Unix Hater's Handbook*. IDG Books. 1994. ISBN 1-56884-203-1.

Available on the Web [http://research.microsoft.com/~daniel/unix-haters.html].

[Gentner-Nielsen] *Communications of the ACM*. Association for Computing Machinery. Don Gentner and Jacob Nielsen. “The Anti-Mac Interface”. August 1996.

Available on the Web [http://www.acm.org/cacm/AUG96/antimac.htm].

[Gettys] Jim Gettys. *The Two-Edged Sword*. 1998.

Available on the Web [http://freshmeat.net/articles/view/122/].

[Glickstein] Bob Glickstein. *Writing GNU Emacs Extensions*. O’Reilly & Associates. 1997. ISBN 1-56592-261-1.

[Graham] Paul Graham. *A Plan for Spam*.

Available on the Web [http://www.paulgraham.com/spam.html].

[Harold-Means] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*. 2nd Edition. O’Reilly & Associates. 2002. ISBN 0-596-00292-0.

[Hatton97] *IEEE Software*. Les Hatton. “Re-examining the Defect-Density versus Component Size Distribution”. March/April 1997.

Available on the Web [http://www.cs.ukc.ac.uk/people/staff/lh8/pubs/pubis697/Ubend_IS697.pdf.gz].

[Hatton98] *IEEE Software*. Les Hatton. “Does OO Sync with the Way We Think?”. 15. (3).

Available on the Web [http://www.cs.ukc.ac.uk/people/staff/lh8/pubs/pubis698/OO_IS698.pdf.gz].

[Hauben] Ronda Hauben. *History of UNIX*.

Available on the Web [http://www.dei.isep.ipp.pt/docs/unix.html].

[Heller] Steve Heller. *C++: A Dialog*. Programming with the C++ Standard Library. Prentice-Hall. 2003. ISBN 0-13-009402-1.

[Hunt-Thomas] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley. 2000. ISBN 0-201-61622-X.

[Kernighan95] Brian Kernighan. *Experience with Tcl/Tk for Scientific and Engineering Visualization*. USENIX Association Tcl/Tk Workshop Proceedings. 1995.

Available on the Web [http://www.usenix.org/publications/library/proceedings/tcl95/full_papers/kernighan.txt].

[Kernighan-Pike84] Brian Kernighan and Rob Pike. *The Unix Programming Environment*. Prentice-Hall. 1984. ISBN 0-13-937681-X.

[Kernighan-Pike99] Brian Kernighan and Rob Pike. *The Practice of Programming*. 1999. ISBN 0-201-61586-X. Addison-Wesley.

An excellent treatise on writing high-quality programs, surely destined to become a classic of the field.

[Kernighan-Plauger] Brian Kernighan and P. J. Plauger. *Software Tools*. Addison-Wesley. 1976. ISBN 201-03669-X.

[Kernighan-Ritchie] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. 2nd Edition. Prentice-Hall Software Series. 1988. ISBN 0-13-110362-8.

[Lampson] *ACM Operating Systems Review*. Association for Computing Machinery. Butler Lampson. “Hints for Computer System Design”. October 1983.

Available on the Web [http://research.microsoft.com/~lampson/33-Hints/WebPage.html].

[Lapin] J. E. Lapin. *Portable C and Unix Systems Programming*. Prentice-Hall. 1987. ISBN 0-13-686494-5.

[Leonard] Andrew Leonard. *BSD Unix: Power to the People, from the Code*. 2000.

Available on the Web [http://dir.salon.com/tech/fsp/2000/05/16/chapter_2_part_one/index.html].

[Levy] Steven Levy. *Hackers: Heroes of the Computer Revolution*. Anchor/Doubleday. 1984. ISBN 0-385-19195-2.

Available on the Web [http://www.stanford.edu/group/mmdd/SiliconValley/Levy/Hackers.1984.book/contents.html].

[Lewine] Donald Lewine. *POSIX Programmer’s Guide: Writing Portable Unix Programs*. 1992. O’Reilly & Associates. ISBN 0-937175-73-0. 607pp..

[Libes-Ressler] Don Libes and Sandy Ressler. *Life with Unix*. 1989. ISBN 0-13-536657-7. Prentice-Hall.

This book gives a more detailed version of Unix's early history. It's particularly strong for the period 1979–1986.

[Lions] John Lions. *Lions's Commentary on Unix 6th Edition*. 1996. 1-57398-013-7. Peer-To-Peer Communications.

PostScript rendering of Lions's original floats around the Web. This URL may be unstable [http://www.upl.cs.wisc.edu/~epaulson/lionc.ps].

[Loukides-Oram] Mike Loukides and Andy Oram. *Programming with GNU Software*. O'Reilly & Associates. 1996. ISBN 1-56592-112-7.

[Lutz] Mark Lutz. *Programming Python*. O'Reilly & Associates. 1996. ISBN 1-56592-197-6.

[McIlroy78] *The Bell System Technical Journal*. Bell Laboratories. M. D. McIlroy, E. N. Pinson, and B. A. Tague. "Unix Time-Sharing System Forward". 1978. 57 (6, part 2). p. 1902.

[McIlroy91] *Proc. Virginia Computer Users Conference*. Volume 21. M. D. McIlroy. "Unix on My Mind". p. 1-6.

[Miller] *The Psychological Review*. George Miller. "The Magical Number Seven, Plus or Minus Two". Some limits on our capacity for processing information. 1956. 63. pp. 81-97.

Available on the Web [http://www.well.com/user/smalin/miller.html].

[Mumon] Mumon. *The Gateless Gate*.

A good modern translation is available on the Web [http://www.ibiblio.org/zen/cgi-bin/koan-index.pl].

[OpenSources] Sam Ockman and Chris DiBona. *Open Sources: Voices from the Open Source Revolution*. O'Reilly & Associates. 1999. ISBN 1-56592-582-3. 280pp..

Available on the Web [http://www.oreilly.com/catalog/opensources/book/toc.html].

[Oram-Talbot] Andrew Oram and Steve Talbot. *Managing Projects with Make*. O'Reilly & Associates. 1991. ISBN 0-937175-90-0.

[Osterhout94] John Osterhout. *Tcl and the Tk Toolkit*. Addison-Wesley. 1994. ISBN 0-201-63337-X.

[Osterhout96] John Osterhout. *Why Threads Are a Bad Idea (for most purposes)*. 1996.

An invited talk at USENIX 1996. There is no written paper that corresponds to it, but the slide presentation is available on the Web [http://home.pacbell.net/ouster/threads.pdf].

[Padlipsky] Michael Padlipsky. *The Elements of Networking Style*. iUniverse.com. 2000. ISBN 0-595-08879-1.

[Parnas] *Communications of the ACM*. Parnas L. David. “On the Criteria to Be Used in Decomposing Systems into Modules”.

Available on the Web at the ACM Classics page [http://www.acm.org/classics/may96/].

[Pike] Rob Pike. *Notes on Programming in C*.

This document is popular on the Web; a title search is sure to find several copies. Here is one [http://www.lysator.liu.se/c/pikestyle.html].

[Prechelt] Lutz Prechelt. *An Empirical Comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a Search/String-Processing Program [http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=ira/2000/5]*.

[Raskin] Jef Raskin. *The Humane Interface*. Addison-Wesley. 2000. ISBN 0-201-37937-6.

A summary is available on the Web [http://humane.sourceforge.net/humane_interface/summary_of_thi.html].

[Ravenbrook] *The Memory Management Reference*.

Available on the Web [http://www.memorymanagement.org/].

[Raymond96] Eric S. Raymond. *The New Hacker’s Dictionary*. 3rd Edition. 1996. ISBN 0-262-68092-0. MIT Press. 547pp..

Available on the Web at Jargon File Resource Page [http://www.catb.org/~esr/jargon].

[Raymond01] Eric S. Raymond. *The Cathedral and the Bazaar*. 2nd Edition. 1999. ISBN 0-596-00131-2. O’Reilly & Associates. 240pp..

[Reps-Senzaki] Paul Reps and Nyogen Senzaki. *Zen Flesh, Zen Bones*. 1994. Shambhala Publications. ISBN 1-570-62063-6. 285pp..

A superb anthology of Zen primary sources, presented just as they are.

[Ritchie79] Dennis M. Ritchie. *The Evolution of the Unix Time-Sharing System*. 1979.

Available on the Web [http://cm.bell-labs.com/cm/cs/who/dmr/hist.html].

[Ritchie93] Dennis M. Ritchie. *The Development of the C Language*. 1993.

Available on the Web [http://cm.bell-labs.com/cm/cs/who/dmr/chist.html].

[RitchieQED] Dennis M. Ritchie. *An Incomplete History of the QED Text Editor*. 2003.

Available on the Web [http://cm.bell-labs.com/cm/cs/who/dmr/qed.html].

[Ritchie-Thompson] *The Unix Time-Sharing System*. Dennis M. Ritchie and Ken Thompson.

Available on the Web [http://cm.bell-labs.com/cm/cs/who/dmr/cacm.html].

[Saltzer] *ACM Transactions on Computer Systems*. Association for Computing Machinery. James. H. Saltzer, David P. Reed, and David D. Clark. “End-to-End Arguments in System Design”. November 1984.

Available on the Web [http://web.mit.edu/Saltzer/www/publications/endtoend/endtoend.pdf].

[Salus] Peter H. Salus. *A Quarter-Century of Unix*. Addison-Wesley. 1994. ISBN 0-201-54777-5.

An excellent overview of Unix history, explaining many of the design decisions in the words of the people who made them.

[Schaffer-Wolf] Evan Schaffer and Mike Wolf. *The Unix Shell as a Fourth-Generation Language*. 1991.

Available on the Web [http://www.rdb.com/lib/4gl.pdf]. An open-source implementation, *NoSQL*, is available and readily turned up by a Web search.

[Schwartz-Christiansen] Randal Schwartz and Tom Phoenix. *Learning Perl*. 3rd Edition. O’Reilly & Associates. 2001. ISBN 0-596-00132-0.

[Spinellis] *Journal of Systems and Software*. Diomidis Spinellis. “Notable Design Patterns for Domain-Specific Languages”. 56. (1). February 2001. p. 91-99.

Available on the Web [].

[Stallman] Richard M. Stallman. *The GNU Manifesto*.

Available on the Web [http://www.gnu.org/gnu/manifesto.html].

[Stephenson] Neal Stephenson. *In the Beginning Was the Command Line*. 1999.

Available on the Web [http://www.cryptonomicon.com/beginning.html], and also as a trade paperback from Avon Books.

[Stevens90] W. Richard Stevens. *Unix Network Programming*. Prentice-Hall. 1990. ISBN 0-13-949876-1.

The classic on this topic. Note: Some later editions of this book omit coverage of the Version 6 networking facilities like *mx()*.

[Stevens92] W. Richard Stevens. *Advanced Programming in the Unix Environment*. 1992. ISBN 0-201-56317-7. Addison-Wesley.

Stevens’s comprehensive guide to the Unix API. A feast for the experienced programmer or the bright novice, and a worthy companion to *Unix Network Programming*.

[Stroustrup] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley. 1991. ISBN 0-201-53992-6.

[Tanenbaum-VanRenesse] Andrew S. Tanenbaum and Robbert van Renesse. *A Critique of the Remote Procedure Call Paradigm*. EUTECO’88 Proceedings, Participants Edition. 1988. pp. 775-783.

[Tidwell] Doug Tidwell. *XSLT: Mastering XML Transformations*. O’Reilly & Associates. 2001. ISBN 1-596-00053-7.

[Torvalds] Linus Torvalds and David Diamond. *Just for Fun*. The Story of an Accidental Revolutionary. HarperBusiness. 2001. ISBN 0-06-662072-4.

[Vaughan] Gary V. Vaughan, Tom Tromey, and Ian Lance Taylor. *GNU Autoconf, Automake, and Libtool*. New Riders Publishing. 2000. 390 p.. ISBN 1-578-70190-2.

A user's guide to the GNU autoconfiguration tools. Available on the Web [http://sources.redhat.com/autobook/].

[Vo] *Software Practice & Experience*. Kiem-Phong Vo. "The Discipline and Method Architecture for Reusable Libraries". 2000. 30. p. 107-128.

Available on the Web [http://www.research.att.com/sw/tools/vcodex/dm-spe.ps].

[Wall2000] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. 3rd Edition. O'Reilly & Associates. 2000. ISBN 0-596-00027-8.

[Welch] Brent Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall. 1999. ISBN 0-13-022028-0.

[Williams] Sam Williams. *Free as in Freedom*. O'Reilly & Associates. 2002. ISBN 0-596-00287-4.

Available on the Web [http://www.oreilly.com/openbook/freedom/index.html].

[Yourdon] Edward Yourdon. *Death March*. The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects. Prentice-Hall. 1997. ISBN 0-137-48310-4.

Appendix C. Contributors

Anyone who has attended a USENIX conference in a fancy hotel can tell you that a sentence like “You’re one of those computer people, aren’t you?” is roughly equivalent to “Look, another amazingly mobile form of slime mold!” in the mouth of a hotel cocktail waitress.

--

<author>ElizabethZwicky</author>

Ken Arnold was part of the group that created the 4BSD Unix releases. He wrote the original curses(3) library and was one of the authors of the original rogue(6) game. He is a co-author of the *Java Reference Manual*, and one of the leading experts on Java and OO techniques.

Steven M. Bellovin created Usenet (with Tom Truscott and Jim Ellis) while at University of North Carolina in 1979. In 1982 he joined AT&T Bell Laboratories, where he has done pioneering research in security, cryptography, and networking on Unix systems and elsewhere. He is an active member of the Internet Engineering Task Force, and a member of the National Academy of Engineering.

Stuart Feldman was a member of the Bell Labs Unix development group. He wrote make(1) and f77(1). He is now the vice-president in charge of computing research at IBM.

Jim Gettys was, with Bob Scheifler and Keith Packard, one of the principal architects of the X windowing system in the late 1980s. He wrote much of the X library, the X license, and articulated the “mechanism, not policy” central credo of the X design.

Steve Johnson wrote yacc(1) and then used it to write the Portable C Compiler, which replaced the original DMR C and became the ancestor of most later Unix C compilers.

Brian Kernighan has been the Unix community’s single most articulate exponent of good style. He has authored or coauthored several books that are indispensable classics of the tradition, including *The Practice of Programming*, *The C Programming Language*, *The Unix Programming Environment*. While at Bell Labs, he coauthored the awk(1) language and had a major hand in the development of the troff family of tools, including eqn(1) (co-written with Lorinda Cherry), pic(1), and grap(1) (Jon Bentley).

David Korn wrote the Korn shell, the stylistic ancestor of almost all modern Unix shell designs. He created UWIN, a UNIX emulator for those that are forced to use Windows. David has also done research in the design of file systems and tools for source-code portability.

Mike Lesk was part of the original Unix crew at Bell Labs. Among other contributions, he wrote the *ms* macro package, the tbl(1) and refer(1) tools for word processing, the lex(1) lexical-analyzer generator, and *UUCP* (Unix-to-Unix copy program).

Doug McIlroy headed the research group at Bell Labs where Unix was born and invented the Unix pipe. He wrote spell(1), diff(1), sort(1), join(1), tr(1), and other classic Unix tools, and helped define the traditional style of Unix documentation. He has also done pioneering work in storage-allocation algorithms, computer security, and theorem-proving.

Marshall Kirk McKusick implemented the 4.2BSD fast file system and was the Research Computer Scientist at the Berkeley Computer Systems Research Group (CSRG) overseeing the development and release of 4.3BSD and 4.4BSD.

Keith Packard was a major contributor to the original X11 code. During a second phase of involvement beginning in 1999, Keith rewrote the X rendering code, producing a more powerful but dramatically smaller implementation suitable for handheld computers and PDAs.

Eric S. Raymond has been writing Unix software since 1982. In 1991 he edited *The New Hacker's Dictionary*, and has since been studying the Unix community and the Internet hacker culture from a historical and anthropological perspective. In 1997 that study produced *The Cathedral and the Bazaar*, that helped (re)define and energize the open-source movement. He presently maintains more than thirty open-source software projects and about a dozen key FAQ documents.

Henry Spencer was a leader among the first wave of programmers to embrace Unix when it escaped from Bell Labs in the mid-1970s. His contributions included the public-domain getopt(3), the first open-source string library, and an open-source regular-expression engine which found use in 4.4BSD and as the reference for the POSIX standard. He is also a noted expert on the arcana of C. He was a coauthor of C News, and has for many years been a voice of reason on Usenet and one of its most respected regulars.

Ken Thompson invented Unix.

Appendix D. Rootless Root

The Unix Koans of Master Foo

無根的根

Editor’s Introduction

The discovery of the collection of koans known as the *Rootless Root*, apparently preserved for decades in the dry upper air of the Western Mountains, has ignited great controversy in scholarly circles. Are these authentic documents shedding new light on the teaching of the early Unix patriarchs? Or are they clever pastiches from a later age, commanding the authority of semi-mythical figures such as the Patriarchs Thompson, Ritchie and McIlroy for doctrines which evolved closer to our own era?

It is impossible to say for certain. All sides in the dispute have made much of an alleged similarity to that venerable classic, *The Tao of Programming*.[165] But *Rootless Root* is quite different in tone and style from the loose, poetic anecdotes of the James translation, focused as it is on the remarkable and enigmatic figure of Master Foo.

It would be more apposite to seek parallels in the *AI Koans*;[166] indeed, there are textual clues that the author of the *Rootless Root* may have redacted certain versions of the *AI Koans*. We are also on much firmer ground in seeking connections with the Loginataka;[167] indeed, it is quite possible that the unknown authors of *Rootless Root* and of *Loginataka* were one and the same person, possibly a student of Master Foo himself.

[165] *The Tao of Programming* is available on the Web [http://www.canonical.org/~kragen/tao-of-programming.html].
[166] The *AI Koans* are available on the Web [http://www.catb.org/~esr/jargon/html/Some-AI-Koans.html].
[167] The *Loginataka* is available on the Web [http://www.catb.org/~esr/faqs/loginataka.html].

Mention should also be made of the *Tales of Zen Master Greg*,[168] though the Nine Inch Nails references have cast some doubt on their antiquity and it is thus unlikely that they influenced *Rootless Root*.

That the title of the work was intended as a reference to the Zen classic *Gateless Gate*[169] of Mumon, we can say with fair confidence. There are echoes of Mumon in several of the koans.

There is considerable dispute over whether Master Foo should be assigned to the Eastern (New Jersey) School, or the Western School that grew out of the Patriarch Thompson's epochal early journey to Berkeley. If this question has not been settled, it is perhaps because we cannot even establish that Master Foo ever existed! He might merely be a composite of a group of teachers, or of an entire dharma lineage.

Even supposing the legend of Master Foo accreted around the teaching of some a single person, what of his favored student Nubi? Nubi has all the earmarks of a stock figure, the perfect disciple. One is reminded of the tales surrounding the Buddha's favorite follower Ananda. It seems likely that there was a historical Ananda, but no trace of his actual personality has survived the euhemerizing process by which the life of the Buddha was polished into timeless myth.

In the end, all we can do is take these teaching stories on their own terms, and extract what kernels of wisdom may be found there.

The redaction of the *Rootless Root* is a work in progress, as the source materials present many difficulties in reconstruction and interpretation. Future versions may include more stories as these difficulties are overcome.

Master Foo and the Ten Thousand Lines

Master Foo once said to a visiting programmer: "There is more Unix-nature in one line of shell script than there is in ten thousand lines of C".

The programmer, who was very proud of his mastery of C, said: "How can this be? C is the language in which the very kernel of Unix is implemented!"

Master Foo replied: "That is so. Nevertheless, there is more Unix-nature in one line of shell script than there is in ten thousand lines of C".

[168]The *Tales of Zen Master Greg* are available on the Web [http://www.gu.uwa.edu.au/users/greg/].
[169]The *Gateless Gate* is available on the Web [http://www.ibiblio.org/zen/cgi-bin/koan-index.pl].

The programmer grew distressed. "But through the C language we experience the enlightenment of the Patriarch Ritchie! We become as one with the operating system and the machine, reaping matchless performance!"

Master Foo replied: "All that you say is true. But there is still more Unix-nature in one line of shell script than there is in ten thousand lines of C".

The programmer scoffed at Master Foo and rose to depart. But Master Foo nodded to his student Nubi, who wrote a line of shell script on a nearby whiteboard, and said: "Master programmer, consider this pipeline. Implemented in pure C, would it not span ten thousand lines?"

The programmer muttered through his beard, contemplating what Nubi had written. Finally he agreed that it was so.

"And how many hours would you require to implement and debug that C program?" asked Nubi.

"Many", admitted the visiting programmer. "But only a fool would spend the time to do that when so many more worthy tasks await him".

"And who better understands the Unix-nature?" Master Foo asked. "Is it he who writes the ten thousand lines, or he who, perceiving the emptiness of the task, gains merit by not coding?"

Upon hearing this, the programmer was enlightened.

Master Foo and the Script Kiddie

A stranger from the land of Woot came to Master Foo as he was eating the morning meal with his students.

"I hear y00 are very l33t", he said. "Pl33z teach m3 all y00 know".

Master Foo's students looked at each other, confused by the stranger's barbarous language. Master Foo just smiled and replied: "You wish to learn the Way of Unix?"

"I want to b3 a wizard hax0r", the stranger replied, "and 0wn ever3one's b0xen".

"I do not teach that Way", replied Master Foo.

The stranger grew agitated. “D00d, y00 r nothing but a p0ser”, he said. “If y00 n00 anything, y00 wud t33ch m3”.

“There is a path”, said Master Foo, “that might bring you to wisdom”. The master scribbled an IP address on a piece of paper. “Cracking this box should pose you little difficulty, as its guardians are incompetent. Return and tell me what you find”.

The stranger bowed and left. Master Foo finished his meal.

Days passed, then months. The stranger was forgotten.

Years later, the stranger from the land of Woot returned.

“Damn you!” he said, “I cracked that box, and it was easy like you said. But I got busted by the FBI and thrown in jail”.

“Good”, said Master Foo. “You are ready for the next lesson”. He scribbled an IP address on another piece of paper and handed it to the stranger.

“Are you *crazy*?” the stranger yelled. “After what I’ve been through, I’m never going to break into a computer again!”

Master Foo smiled. “Here”, he said, “is the beginning of wisdom”.

On hearing this, the stranger was enlightened.

Master Foo Discourses on the Two Paths

Master Foo instructed his students:

“There is a line of dharma teaching, exemplified by the Patriarch McIlroy’s mantra ‘Do one thing well’, which emphasizes that software partakes of the Unix way when it has simple and consistent behavior, with properties that can be readily modeled by the mind of the user and used by other programs”.

“But there is another line of dharma teaching, exemplified by the Patriarch Thompson’s great mantra ‘When in doubt, use brute force’, and various sutras on the value of getting 90% of cases right *now*, rather than 100% *later*, which emphasizes robustness and simplicity of implementation”.

“Now tell me: which programs have the Unix nature?”

After a silence, Nubi observed:

“Master, these teachings may conflict”.

“A simple implementation is likely to lack logic for edge cases, such as resource exhaustion, or failure to close a race window, or a timeout during an uncompleted transaction”.

“When such edge cases occur, the behavior of the software will become irregular and difficult. Surely this is not the Way of Unix?”

Master Foo nodded in agreement.

“On the other hand, it is well known that fancy algorithms are brittle. Further, each attempt to cover an edge case tends to interact with both the program’s central algorithms and the code covering other edge cases”.

“Thus, attempts to cover all edge cases in advance, guaranteeing ‘simplicity of description’, may in fact produce code that is overcomplicated and brittle or which, plagued by bugs, never ships at all. Surely this is not the Way of Unix?”

Master Foo nodded in agreement.

“What, then, is the proper dharma path?” asked Nubi.

The master spoke:

“When the eagle flies, does it forget that its feet have touched the ground? When the tiger lands upon its prey, does it forget its moment in the air? Three pounds of VAX!”

On hearing this, Nubi was enlightened.

Master Foo and the Methodologist

When Master Foo and his student Nubi journeyed among the sacred sites, it was the Master’s custom in the evenings to offer public instruction to Unix neophytes of the towns and villages in which they stopped for the night.

On one such occasion, a methodologist was among those who gathered to listen.

“If you do not repeatedly profile your code for hot spots while tuning, you will be like a fisherman who casts his net in an empty lake”, said Master Foo.

“Is it not, then, also true”, said the methodology consultant, “that if you do not continually measure your productivity while managing resources, you will be like a fisherman who casts his net in an empty lake?”

“I once came upon a fisherman who just at that moment let his net fall in the lake on which his boat was floating”, said Master Foo. “He scrabbled around in the bottom of his boat for quite a while looking for it”.

“But”, said the methodologist, “if he had dropped his net in the lake, why was he looking in the boat?”

“Because he could not swim”, replied Master Foo.

Upon hearing this, the methodologist was enlightened.

Master Foo Discourses on the Graphical User Interface

One evening, Master Foo and Nubi attended a gathering of programmers who had met to learn from each other. One of the programmers asked Nubi to what school he and his master belonged. Upon being told they were followers of the Great Way of Unix, the programmer grew scornful.

“The command-line tools of Unix are crude and backward”, he scoffed. “Modern, properly designed operating systems do everything through a graphical user interface”.

Master Foo said nothing, but pointed at the moon. A nearby dog began to bark at the master’s hand.

“I don’t understand you!” said the programmer.

Master Foo remained silent, and pointed at an image of the Buddha. Then he pointed at a window.

“What are you trying to tell me?” asked the programmer.

Master Foo pointed at the programmer’s head. Then he pointed at a rock.

"Why can't you make yourself clear?" demanded the programmer.

Master Foo frowned thoughtfully, tapped the the programmer twice on the nose, and dropped him in a nearby trashcan.

As the programmer was attempting to extricate himself from the garbage, the dog wandered over and piddled on him.

At that moment, the programmer achieved enlightenment.

Master Foo and the Unix Zealot

A Unix zealot, having heard that Master Foo was wise in the Great Way, came to him for instruction. Master Foo said to him:

"When the Patriarch Thompson invented Unix, he did not understand it. Then he gained in understanding, and no longer invented it".

"When the Patriarch McIlroy invented the pipe, he knew that it would transform software, but did not know that it would transform mind".

"When the Patriarch Ritchie invented C, he condemned programmers to a thousand hells of buffer overruns, heap corruption, and stale-pointer bugs".

"Truly, the Patriarchs were blind and foolish!"

The zealot was greatly angered by the Master's words.

"These enlightened ones", he protested. "gave us the Great Way of Unix. Surely, if we mock them we will lose merit and be reborn as beasts or MCSEs".

"Is your code ever completely without stain and flaw?" demanded Master Foo.

"No", admitted the zealot, "no man's is".

"The wisdom of the Patriarchs", said Master Foo, "was that they *knew* they were fools".

Upon hearing this, the zealot was enlightened.

Master Foo Discourses on the Unix-Nature

A student said to Master Foo: “We are told that the firm called SCO holds true dominion over Unix”.

Master Foo nodded.

The student continued, “Yet we are also told that the firm called OpenGroup also holds true dominion over Unix”.

Master Foo nodded.

“How can this be?” asked the student.

Master Foo replied:

“SCO indeed has dominion over the code of Unix, but the code of Unix is not Unix. OpenGroup indeed has dominion over the name of Unix, but the name of Unix is not Unix”.

“What, then, is the Unix-nature?” asked the student.

Master Foo replied:

“Not code. Not name. Not mind. Not things. Always changing, yet never changing”.

“The Unix-nature is simple and empty. Because it is simple and empty, it is more powerful than a typhoon”.

“Moving in accordance with the law of nature, it unfolds inexorably in the minds of programmers, assimilating designs to its own nature. All software that would compete with it must become like to it; empty, empty, profoundly empty, perfectly void, hail!”

Upon hearing this, the student was enlightened.

Master Foo and the End User

On another occasion when Master Foo gave public instruction, an end user, having heard tales of the Master’s wisdom, came to him for guidance.

He bowed three times to Master Foo. “I wish to learn the Great Way of Unix”, he said “but the command line confuses me”.

Some of the onlooking neophytes began to mock the end user, calling him “clueless” and saying that the Way of Unix is only for those of discipline and intelligence.

The Master held up a hand for silence, and called the most obstreperous of the neophytes who had mocked forward, to where he and the end user sat.

“Tell me”, he asked the neophyte, “of the code you have written and the works of design you have uttered”.

The neophyte began to stammer out a reply, but fell silent.

Master Foo turned to the end-user. “Tell me”, he inquired, “why do you seek the Way?”

“I am discontent with the software I see around me”, the end user replied. “It neither performs reliably nor pleases the eye and hand. Having heard that the Unix way, though difficult, is superior, I seek to cast aside all snares and delusions”.

“And what do you do in the world”, asked Master Foo, “that you must strive with software?”

“I am a builder”, the end user replied, “Many of the houses of this town were made under my chop”.

Master Foo turned back to the neophyte. “The housecat may mock the tiger”, said the master, “but doing so will not make his purr into a roar”.

Upon hearing this, the neophyte was enlightened.

Colophon

No proprietary software was used during the composition of this book. Drafts were typeset from XML-DocBook master files created with *GNU Emacs*. PostScript generation was performed with Tim Waugh's *xmlto*, Norman Walsh's XSL stylesheets, Daniel Veillard's *xsltproc*, Sebastian Rahtz's PassiveTeX macros, the TeTeX distribution of Donald Knuth's *TeX* typesetter, and Thomas Rokicki's *dvips* postprocessor. All the diagrams were composed by the author using *pic2graph* driving *gpic* and *grap2graph* driving Ted Faber's *grap* implementation (*grap2graph* was written by the author for this project and is now part of the *groff* distribution). The entire toolchain was hosted by stock Red Hat Linux.

The cover art is a composite of two images from the original *Zen Comics* by Ioanna Salajan. It was adapted and colored mainly by Jerry Votta, but the author and GIMP had a hand in the work.